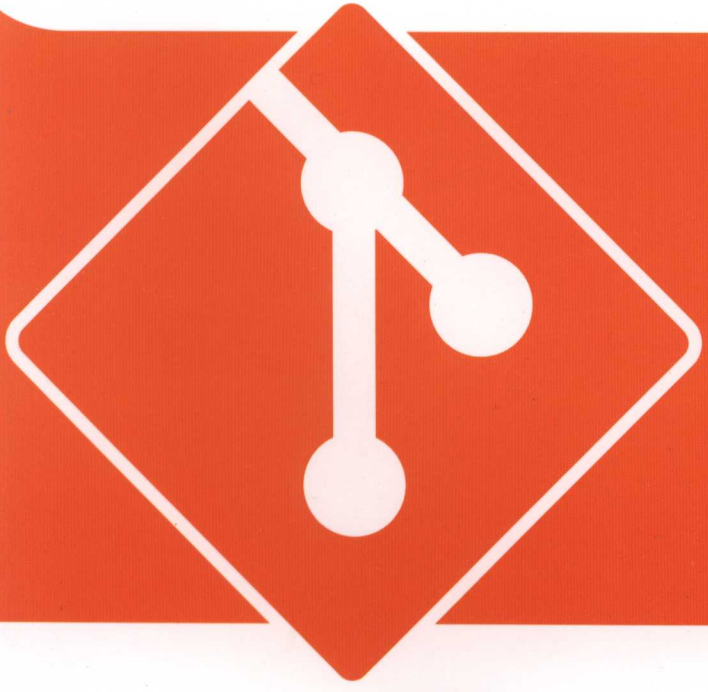
版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

GOTOP

完全学会 Git GitHub Git Server 的 24 堂课



- 针对项目开发人员：以最有效率的方式，引导你在实践中完整地学习Git的操作技巧和应用。
- 针对项目管理者：在实例中学习如何查看项目开发的各种统计图，以便随时掌握项目开发的状况。
- 针对MIS人员：提供建立Git Server的多种方法，自行选用和规划最适合自己实际运用的方法。

清華大學出版社

孙宏明 著

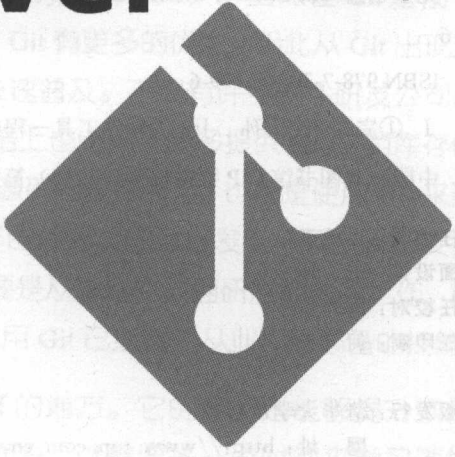
作者简介

孙宏明

- 台湾成功大学信息工程研究所博士
- 台湾友立信息公司主任研究员
- 台湾开南大学资管系副教授
- 台北城市科技大学、台湾元培科技大学、台湾侨光科技大学等大专院校Android App开发研习课程讲师

完全学会 Git GitHub Git Server 的 24 堂课

孙宏明 著



清華大學出版社
北京

北京市版权局著作权合同登记号：图字：01-2016-2268
本书为碁峰资讯股份有限公司授权出版发行的中文简体字版本。

内 容 简 介

本书以教科书的方式，循序渐进地向软件开发人员或软件开发的项目管理人员，讲述如何快速掌握分布式版本控制系统 Git 的应用方法，让繁琐复杂的项目开发的版本控制变得高效且轻松自如。书中细致入微地介绍了如何使用设置文件和操作 Git 文档库，以及建立项目开发的分支、合并分支和解决冲突的操作；在具备扎实的基础之后，介绍了 Git GUI 程序用法，以及 Git 在团队开发模式中应用的问题；讲述如何与全球的项目开发人员进行交流，提升程序开发的经验和能力；介绍了 GitHub、Bitbucket、GitLab 等提供 Git 服务的网站；本书最后的重点为介绍架设 Git Server 的几种方法，让每一个项目开发团队，可以按照自己的软硬件环境，选择适合的方式来控管项目开发。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目（CIP）数据

完全学会 Git, GitHub, Git Server 的 24 堂课/孙宏明著. —北京：清华大学出版社，2016

ISBN 978-7-302-43779-6

Ⅰ. ①完… Ⅱ. ①孙… Ⅲ. ①软件工具－程序设计Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字（2016）第 100121 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：何　芊

出版发行：清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦 A 座　　**邮　编**：100084
社总机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印　装　者：三河市春园印刷有限公司
经　　销：全国新华书店
开　　本：170mm×230mm　　**印　张**：17　　**字　数**：237 千字
版　　次：2016 年 6 月第 1 版　　**印　次**：2016 年 6 月第1 次印刷
印　　数：1～3000
定　　价：49.00 元

产品编号：069428-01

序

在现今的信息产业中，软件扮演的角色越来越重要。如果公司想要在信息产业界中占有一席之地，拥有强大的软件研发实力是必要的条件。以现阶段软件产业的生态来说，能够熟练使用工具软件来提升软件研发的效率，是影响软件实力的关键因素之一，而 Git 就是能够有效提升软件研发效率的重要工具。

Git 青出于蓝、更胜于蓝

Git 最初是由 Linux kernel 的作者 Torvalds 先生所开发，用来管理 Linux kernel 项目的工具。它的功能是负责项目的版本控制，我们称它为“版本控制系统”（Version Control System, VCS）。虽然在 Git 之前，已经有许多 VCS。但是相比之下，Git 有更多的优点。因此从 Git 出现之后，就以“后来者居上”的态势快速普及。不仅有许多软件研发公司改用 Git 来管理和控制软件项目，网络上也出现了许多提供 Git 文档库存储服务的网站。而且绝大多数的开放源码的程序项目，也都是使用 Git 来进行版本控制的管理。为了顺应这样的趋势，软件开发工具现在也都支持用 Git 来进行版本控制。因此，只要是从事软件项目研发相关的工作，或是未来即将踏入这个领域，学会使用 Git 已经成为从业人员必备的技能。

Git 和其他 VCS 有许多不一样的地方。它的优点是能够提升软件开发团队的工作效率，但是相对而言，就需要多一点时间来学习更好地使用它。初次使用 Git 的人通常会有似懂非懂的感觉，仿照别人的操作虽然很容易，可是究竟这样操作会得到什么结果，或是在不同的情况下该如何处理，这一切都会觉得懵懵懂懂。如果是和软件研发团队一起工作，每一次操作 Git 更是胆颤心惊，害怕一不小心，毁了整个团队的心血结晶，成为大家的罪人！

其实 Git 的核心是保护我们的文件，因此它工作的最高指导原则，就是不会遗失我们的数据，除非我们执行删除文件的指令。因此，

Git 的安全性是无庸置疑的，只要了解它的运行方式，熟悉各种情况下的操作方法，就能够让它发挥最大的效能。

Git 的学习之道

本书以循序渐进的方式，学习 Git 的功能和用法。书中以实际的操作范例，让读者亲身体验 Git 工作的细节，并且掌握每一个指令的效果以及每一个选项的使用时机。虽然网络上有许多采用 GUI（图形用户界面）的 Git 程序，但是笔者不建议初学者一开始就使用这类型的程序。因为它的操作界面过于复杂，初学者在还没有完全了解的情况下，常常会觉得疑惑，反而容易让学习失去重点。相比之下，输入 Git 指令的操作方式虽然比较麻烦一点，但是却比较清楚，初学者可以完全理解每一个操作和选项所达成的结果。等到建立好 Git 指令操作的基础之后，再来使用 GUI 界面的 Git 程序，就会水到渠成，轻而易举。

本书的内容安排先从 Git 的单机操作开始，让读者先了解 Git 如何工作，并且熟悉配置文件和文件库的基本操作方法。接着学习如何建立分支、合并分支和解决冲突。等到基础完备之后，再来介绍 GUI 操作界面的 Git 程序，让读者体验不同的操作风格。接着学习 Git 在团队开发模式的应用，以及介绍网络上提供 Git 文件库存储服务的网站。让读者了解如何使用这些资源，辅助程序项目的开发。使用这些网站的另一个好处是可以藉此参与开放源码的程序项目，提升自己程序开发的经验和能力。本书的最后部分是介绍架设 Git Server 的几种方法，让每一个项目开发团队，可以按照自己的软硬件环境，选择适合的方式来管理和控制项目。

相信通过本书精心编排的学习单元，能够顺利地引导初学者，一步一步地建立完整的操作观念和技巧，并且培养在不同情况下的应用能力。一旦读者能够完全掌握 Git 之后，它必定成为你或是你的团队开发程序项目的最佳助手！

孙宏明

改编说明

如果你是一位软件开发人员或者是参与软件开发的项目管理人员，并且还苦恼于如何高效且轻松地管理和控制软件项目的版本控制，那么赶紧学习 Git 吧！Git 是一款免费、开源的分布式版本控制系统，也是世界上目前最先进的分布式版本控制系统之一。

Git 是 Linux 内核的发明人及该计划的合作者 Linus Torvalds 为了帮助 Linux 内核开发和维护，而开发的一款开放源码的版本控制软件。Git 具有极大的适用弹性，可运用于任何规模的软件项目的版本管理——不但适用于单个人开发的很小项目，而且也适用于众多人开发的大型项目。既可以用于本地合作开发的软件项目的版本控制和管理，也可以用于跨地域合作开发的软件项目的版本控制和管理。

虽然 Git 开发的最初目的是用于辅助 Linux 内核开发的过程，但是越来越多的自由软件项目和商业软件项目都使用了 Git 进行灵活、高效和快捷的版本控制。

Git 有这么多优点，但是因为中文资料和中文书籍相对较少，也不是很系统，所以初学者学习周期相对而言会比较长，学习起来也不是很系统。本书非常系统地介绍了 Git 及其在项目开发中的实践运用，对于想快速学习 Git 以及系统掌握 Git 的人员确实是一本不错的教科书和参考书。

本书适合用于软件开发和软件项目管理人员进行自学，也适合软件开发公司作为公司内部的专业培训教材。

赵军

目 录

第一部分 Git 的基本概念和操作

第 1 课 Git、谁与争锋 2

1-1 安装和使用 Git 3

1-2 了解 Git 的工作方式 9

第 2 课 Git 配置文件的妙用 11

2-1 “git config”指令的用法 12

2-2 修改默认的文本编辑器和文件对比程序 15

第 3 课 把文件存入 Git 文档库 20

3-1 排除不需要加入文档库的文件 20

3-2 控制 Commit 25

3-3 查看 Commit 节点 27

第 4 课 比较文件的差异和从 Git 文档库取回文件 31

4-1 从 Git 文档库中取出文件 33

4-2 使用“git mv”指令改变文件或是文件夹的名称 36

4-3 暂存当前文件夹的文件状态 37

4-4 清理 Git 文档库 38

第 5 课 Git 程序项目管理的实践演练 40

第 6 课　获取 Git 文档库统计数据和绘制统计图表 45

6-1　获取 Git 文档库的统计数据 46
6-2　使用 GitStats 绘制统计图表 49

第二部分　建立分支、合并和解决冲突

第 7 课　程序项目的分支（Branch） 56

7-1　解决 Detached HEAD 的问题 63

第 8 课　合并程序项目的分支和解决冲突 67

8-1　解决合并分支时发生的冲突 71
8-2　设置 Merge Tool 解决合并分支时发生的冲突 75
8-3　使用 Cherry-Pick 指令运用指定的 Commit 节点版本 77
8-4　处理 Detached HEAD 状态和 Revert Commit 节点 79

第 9 课　使用 Rebase 指令更新分支的起始点 81

9-1　Rebase 指令的用法 84
9-2　执行 Rebase 之后想要反悔怎么办？ 86

第 10 课　程序项目的分支和合并的实践演练 88

10-1　程序项目开发导入 Git 的策略 88
10-2　结合 Git 与程序项目开发 91

第三部分　Git 程序的图形操作介面

第 11 课　Git GUI 程序的基本功能 99

11-1　Git GUI 程序的基本操作 101

11-2 Git GUI 程序和其他程序搭配使用 104

第 12 课 使用 Git GUI 程序创建分支和合并 108

第 13 课 SmartGit 程序操作介绍 114

13-1 SmartGit 程序的基本操作 116
13-2 SmartGit 程序的分支、Rebase 和合并 120

第 14 课 SourceTree 程序操作介绍 127

14-1 SourceTree 程序的基本操作 128
14-2 SourceTree 程序的分支、Rebase 和合并 131

第 15 课 TortoiseGit 程序操作介绍 138

15-1 TortoiseGit 程序的基本操作 139
15-2 TortoiseGit 程序的分支、Rebase 和合并 145

第四部分 远程 Git 文档库和团队开关模式

第 16 课 “远程 Git 文档库”的基础操作 154

16-1 “远程 Git 文档库”的功能 155
16-2 创建“远程 Git 文档库” 156
16-3 从“远程 Git 文档库”复制出“本地 Git 文档库” 158
16-4 “本地 Git 文档库”和“远程 Git 文档库”的数据同步 159

第 17 课 “远程 Git 文档库”的高级操作 167

第 18 课 GitHub 让程序项目飞上云端 172

18-1 注册和设置 GitHub 173

18-2 使用 GitHub 的“远程 Git 文档库” 177
18-3 安装和使用 GitHub for Windows 182

第 19 课 Fork 让 Git 文档库分家后再合并 189

19-1 Fork 和 Rebase Git 文档库 192
19-2 创建 Pull Request 报告我们的更新 195

第 20 课 Bitbucket 比 GitHub 更好用 199

20-1 注册和设置 Bitbucket 200
20-2 使用 Bitbucket 的“远程 Git 文档库” 203
20-3 使用 SourceTree 程序操作“远程 Git 文档库” 205
20-4 执行 Fork 和 Pull Request 208

第 21 课 GitLab 完全免费再加送整个系统 211

21-1 自己架设 GitLab 网站 219

第五部分 架设 Git Server

第 22 课 使用 Windows 共享文件夹或是 Git Daemon 作为 Git Server 225

22-1 把 Windows“共享文件夹”作为“远程 Git 文档库” 226
22-2 使用 Git Daemon 创建 Git Server 227

第 23 课 使用 HTTP/HTTPS 访问 Git Server 231

23-1 使用 HTTP 访问 Git Server 232
23-2 使用 HTTPS 访问 Git Server 236

第 24 课　使用 SSH 认证和加密的 Git Server.....239

24-1　SSH 的工作原理 240

24-2　安装和设置 Copssh Server.....241

附录　Git 常用指令一览表.....249

第一部分　Git 的基本概念和操作

第 1 课　Git、谁与争锋

第 2 课　Git 配置文件的妙用

第 3 课　把文件存入 Git 文档库

第 4 课　比较文件的差异和从 Git 文档库取回文件

第 5 课　Git 程序项目管理的实践演练

第 6 课　获取 Git 文档库统计数据和绘制统计图表



第 1 课

Git、谁与争锋

“版本控制系统”（Version Control System, VCS）是程序代码管理软件的通称，是用来保存程序文件的修改记录以及历史版本，以便日后查看或是使用。VCS 已经有数十年的发展历史，最早期的系统是采用集中管理和控制的方式。如果要修改某一个程序文件，必须先将它锁定，然后取出修改。在完成修改和回传之前，其他人都不能更改这个程序文件。这种方式可以有效避免“冲突”（conflict）的发生（也就是防止不同人同时修改同一段程序代码所造成混淆的情况），但是付出的代价是效率降低。因为如果想要修改的文件正好有别人正在修改，就必须等程序文件回传之后才能动手。如果要修改的人很多，因为互相等待所造成的时间浪费将非常可观。为了解决这个问题，新的 VCS 改成采用分布式的方法。每一个人随时都可以获取任何一个程序文件来进行修改，等到送回 VCS 的时候，再根据需要进行“合并”（merge），Git 就是采用这种分布式技术的系统。

当前市场上占有率最高的 VCS 有二套：Subversion（简称 SVN）和 Git。在 2013 年以前，SVN 的市场占有率维持在 40%以上。但是之后，Git 的使用率开始接近 SVN，至 2014 年以后就超越了 SVN。现在越来越多的软件公司开始使用 Git 管理程序项目，网络上也有专门提供 Git Server 服务的网站，像是 GitHub 和 GitLab。许多开放源码的程序项目，例如 Linux 和 Android，也都是使用 Git 管理程序代码的，在职场上要求具备 Git 应用能力的情况也越来越普遍。如果你的 Git 功力深厚，甚至具备架设和管理 Git Server 的能力，在职场上必定更受青睐。不过在开始学习 Git 之前，让我们先了解一下 Git 是如何诞生的。

Git 的开发者其实就是 Linux 的作者，也就是鼎鼎大名的 Linus Torvalds。一开始的时候，Torvalds 是使用 BitKeeper 软件来管理 Linux 的程序代码，该软件原来是免费使用的，但是后来却要收费。于是 Torvalds 想要更换 VCS，却苦于寻找不到其他的替代方案可以符合开发 Linux 的需求。因为参与开发 Linux 的人数高达上千人，而且分布在世界各地。最后 Torvalds 决定自己打造一个 VCS。经过短短几个星期，第一版的 Git 就正式诞生。Linux 项目从此用 Git 管理，当时有将近七百万行的程序代码！

由于 Git 管理 Linux 项目非常成功，许多公司于是纷纷效仿，开始使用 Git 来管理和控制软件研发项目。这些成功的经验，让 Git 的影响力越来越高，使用率也快速提升。

1-1 安装和使用 Git

每一个操作系统平台都可以找到适合的 Git 安装程序，以下是官方网站的下载网址：

http://git-scm.com/downloads

除了官方的版本，也有其他软件公司将 Git 包装成 GUI 操作界面类型的程序（官方版本主要是使用指令模式进行操作）。虽然 GUI 的图形用户操作画面看起来比较吸引人，但是建议初学者还是先从 Git 的指令模式开始学习，这样才能够清楚地了解 Git 的工作细节。等到具备完整的应用基础以后，再转换到 GUI 界面，自然就能够水到渠成。本书将带领读者从基础的指令操作开始，一直到 GUI 的操作模式以及各种情况的应用，最后还会介绍 Git Server 的架设和管理，让读者能够熟练使用 Git 来管理程序代码，从而提高程序项目的开发效率，现在就让我们开始动手吧。

1 步骤 安装 Git。如果是 Windows 平台，直接运行下载的安装程序。在安装选项设置页面中勾选 Git Bash Here 和 Git GUI Here（参见图 1-1），然后按照说明完成安装。其他操作系统平台请参考网页上的说明来完成安装。

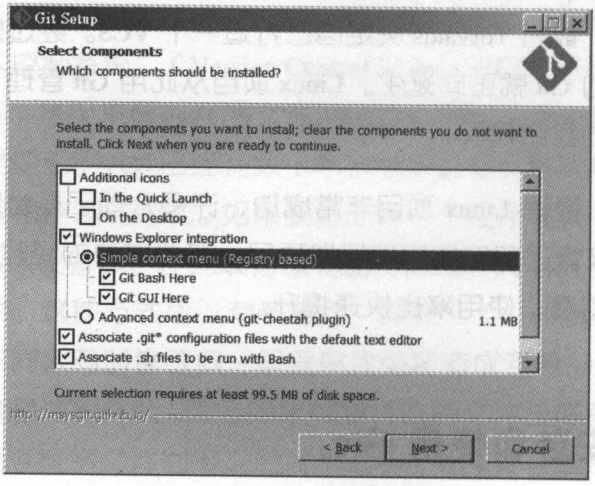


图 1-1　Windows 平台安装 Git 时的选项

2 步骤 安装完毕之后，我们就来体验一下 Git 的操作方式。先创建一个新的文件夹，文件夹名称可以使用中文或是英文。

3步骤 启动 Git Bash 程序，使用 cd 指令切换到前一个步骤创建的文件夹：

```
cd '文件夹路径'
```

如果文件夹路径中包含空格，必须用单引号或是双引号将路径括起来，如以上指令格式。完成之后就会看到如下信息：

```
用户名称@计算机名称 文件夹路径
$
```

4步骤 现在，我们要让 Git 开始管理这个文件夹，请执行以下指令：

```
git init
```

Git 会响应消息“Initialized empty Git repository in 文件夹路径/.git/”，表示已经完成准备工作。

补充说明

1. Git 可以管理任何一个文件夹中的文件和子文件夹，只要在该文件夹中执行“git init”，就可以让 Git 完成管理前的准备工作。Git 会在这个文件夹创建所谓的 repository（我们把它称为“文档库”），里面存储被管理的文件和文件夹内容，包括所有曾经被加入的历史版本。
2. “文档库”其实是名称叫作“.git”的子文件夹，默认它会被隐藏起来，我们可以改变文件夹的查看选项，让隐藏的文件和文件夹也能够正常显示。
3. Git 可以正确处理中文名称的文件和文件夹，只是 Git Bash 程序在显示中文的时候会出现乱码，后续我们会介绍 gitk 图形查看模式，它可以正确显示中文。
4. 单独执行“git”指令会显示辅助说明；执行“git help - a”则显示完整的指令列表；执行“git 指令 - help”（例如“git ini t - help”）则会显示该指令的网页说明文件。
5. 如果指令太长，想要换到下一行继续输入，可以用反斜杠字符“\”结尾，然后按下 Enter 键，继续输入。

5 步骤 现在我们创建一个纯文本文件来测试 Git 如何管理文件，请将这个文本文件取名为 poem.txt，在里面输入一行文字如下（这是唐朝诗人“王维”的《山居秋暝》）：

```
空山新雨后
```

6 步骤 回到 Git Bash 程序，依序执行以下两个指令：

```
git add poem.txt
git status
```

Git 会响应以下信息：

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   poem.txt
```

这些信息告诉我们，现在有一个名称叫作 poem.txt 的新文件将要被送进“文档库”存储。其实这是执行第一个指令，也就是“git add poem.txt”的结果，第二个指令只是显示当前 Git 的状态。

7 步骤 前一个步骤是准备把文件送进 Git 文档库，现在我们要确实把它送进去，请继续执行：

```
git commit -m '这次操作的说明' --author='操作者姓名 <email 邮箱>'
```

Git 会响应以下信息，显示是谁执行了这项工作，以及有几个文件被改动了：

```
[master (root-commit) 57d02a7] first commit
 Author: 操作者姓名 <email 邮箱>   ← 谁执行这项工作
 1 file changed, 1 insertion(+)← 有一个文件被改动了，总共加入一行文字
 create mode 100644 poem.txt    ← 被改动的文件名
```

如果执行 commit 指令以后，想要修改操作说明或是操作者的资料，可以使用“--amend”选项，也就是：

```
git commit --amend -m '新的操作说明' --author='操作者姓名 <email 邮箱>'
```

8 步骤 接下来修改一下 poem.txt 文件的内容，请将它打开，加入第二行文字如下，再将它保存。

```
空山新雨后
天气晚来秋    ← 这是新加入的文字
```

9 步骤 回到 Git Bash 程序，按序执行下列指令，将修改后的 poem.txt 文件送进文档库存储。

```
git add poem.txt
git commit -m '这次操作的说明' --author='操作者姓名 <email 邮箱>'
```

10 步骤 现在让我们看一下当前文档库中有哪些文件，请执行以下指令，启动图形查看模式：

```
gitk
```

屏幕上会显示图 1-2 所示的画面，左上角会有不同颜色的小节点，每一个节点表示执行一次“git commit”指令。节点右边会显示我们输入的操作说明的第一行文字，左下角窗口会显示当前选定节点的完整信息。最上面的节点会多出一个 master 标签，表示这是文件的“主要分支（branch）”。节点会按照时间顺序从上往下排列，最上面是最近一次的 commit。节点窗口右边还有两个窗口，分别显示执行指令的人和时间。右下角的窗口是当前选定的 commit 节点的内容，它有两种查看模式：patch 和 tree。patch 模式会显示和前一个 commit 节点的差异，tree 模式则显示完整的节点内容。我们可以用鼠标单击要查看的文件，左边的窗口会显示它的内容。读者可以在左上角的窗口单击不同的节点，再使用 tree 模式观察 poem.txt 文件

的内容。到当前为止，我们已经把两个不同版本的 poem.txt 文件送进文档库存储。

补充说明

文件可以有一个以上的“分支”，假如有两个人同时想要修改同一个文件，可以在不同的“分支”上进行，稍后我们可以将不同的“分支”合并，后续我们会再详细介绍这种操作模式。

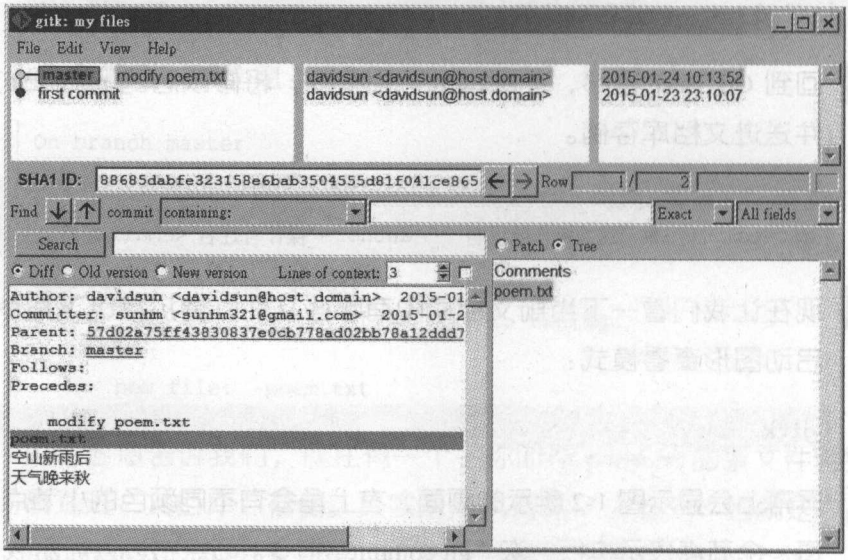


图 1-2　用图形查看模式查看文档库

11 步骤 查看完文档库之后，结束图形操作模式，回到 Git Bash 程序。然后执行以下指令离开 Git Bash。Git 会把当前的执行状态记录在文件夹内，下次再启动 Git Bash 程序，回到这个文件夹的时候，会自动恢复到离开时的状态。

```
exit
```

我们已经完成了第一次 Git 探索之旅，实际体验 Git 的操作方式，也学会了几个指令的用法，或许还有点似懂非懂的感觉，没关系，在下一个小节我们会进一步解释 Git 工作的方式。

1-2 了解 Git 的工作方式

操作 Git 的基本流程就是先修改文件，然后执行“git add”指令。“git add”指令会把文件内容加入 Git 系统的索引。接着就可以执行“git commit”指令，将文件内容存入文档库。于是文档库中就多出一份文件的新版本，我们可以把整个过程用图 1-3 来描述。如果同时要把多个文件送进文档库存储，可以在 git add 指令后面逐一列出这些文件的名称（用空格符隔开），也可以使用通配符“*”，或是执行多次“git add”指令，最后再执行“git commit”。“git add”指令后面也可以指定文件夹名称，这样该文件夹内的文件都会被处理。如果执行“git add .”指令（“.”表示当前这个文件夹），则全部的文件和子文件夹都会被处理。如果想要忽略特定文件，可以在文件夹中创建“.gitignore”文件，后续我们会介绍它的用法。

补充说明

有些 Git 指令执行的时候会显示英文信息，里面可能包含 Git 专用的术语，以下是这些专用术语的说明：

1. working tree：就是当前 Git 正在管理的这个文件夹，因为文件夹是一个树状结构，所以 Git 将它称为 working tree。
2. index：就是前面所说的“Git 系统索引”，把文件内容加入 Git 索引又称为 stage 或是 cache，把文件内容从索引中删除则称为 unstage。
3. repository：就是“Git 文档库”，也就是 Git 存储文件的地方。

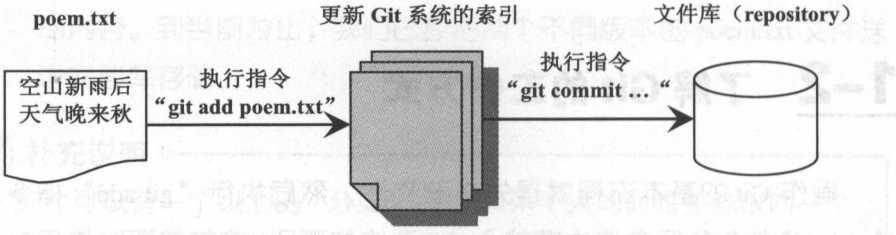


图 1-3　把文件存入 Git 文档库的流程

最后我们再做一个测试来澄清一个容易被初学者误解的现象。

1 步骤 启动 Git Bash 程序，使用 cd 指令切换到刚才的工作文件夹。

2 步骤 打开 poem.txt 文件，加入第三行文字如下：

```
空山新雨后
天气晚来秋
明月松间照　← 这是新加入的文字
```

3 步骤 将修改后的文件内容加入 Git 系统的索引：

```
git add poem.txt
```

4 步骤 再度打开 poem.txt 文件，加入第四行文字：

```
空山新雨后
天气晚来秋
明月松间照
清泉石上流　← 这是新加入的文字
```

5 步骤 最后仿照前面的方式，执行“git commit”指令。

现在请执行“gitk”指令启动图形查看模式，看看文档库中的 poem.txt 文件究竟是三行还是四行文字。结果可能出乎你的意料之外！poem.txt 文件中只有三行文字，最后加入的第四行文字并没有被送进文档库存储。这是因为执行“git add”指令时，只是把当前的文件内容加到 Git 索引。如果之后文件内容又被修改，必须再次执行“git add”指令才会更新 Git 索引，这一点请特别留意。



第 2 课

Git 配置文件的妙用

现在我们已经初步了解 Git 的功能和操作方式。俗话说“工欲善其事，必先利其器”，我们必须知道如何控制 Git 程序的运行，才能够让它发挥最大的功能。每一个软件都有配置文件，它可以决定程序要如何执行，Git 也不例外。Git 有三个不同级别的配置文件，它们有不同的优先权，高优先权文件的设置项会覆盖低优先权文件中相同的设置项。以下按照优先权从高至低按序介绍：

1. 文件夹中“.git”子文件夹内的 config 文件

 这个配置文件具有最高的优先权。也就是说，它的设置会覆盖其他配置文件中相同的设置项。举例来说，如果这个文件中设置 A=B，就算以下两个配置文件设置 A=C 最终生效的还是 A=B。但是这个配置文件只对它所在的文档库有效。

2. 登录账号的 home directory 中的.gitconfig 文件

 只有在前一个配置文件中没有设置的项，这个配置文件的设置才会生效，而且这个配置文件只对此账号登录的用户有效。

3. Git 程序的安装文件夹中的 etc\gitconfig 文件

 只有在前两个配置文件中没有设置的项，这个配置文件的设置才会生效。这是公用的配置文件，它对所有登录账号和所有 Git 文档库都有效。

要显示、添加或是修改配置文件中的设置项，必须使用“git config”指令，下面我们就来学习它的用法。

2-1 “git config”指令的用法

要显示当前 Git 的设置值可以执行下列指令：

```
git config -l
```

这个指令会显示三个配置文件中所有的设置项（参见图 2-1），它的顺序是先显示优先权最低的设置，也就是前面介绍的第三个配置文件中的设置（图 2-1 中标示 A 的部分），再按序显示优先权比较高的设置。图 2-1 中标示 B 的部分是登录账号的 home directory 中的.gitconfig 文件的设置，C 部分是文档库文件夹内“.git”子文件夹中的 config 文件的设置。我们也可以指定显示单一配置文件的内容如下：

```
git config --system -l ← 显示 Git 程序安装文件夹中的 etc\gitconfig 文件内的设置
git config --global -l ← 显示登录账号的 home directory 中的.gitconfig 文件内的设置
```

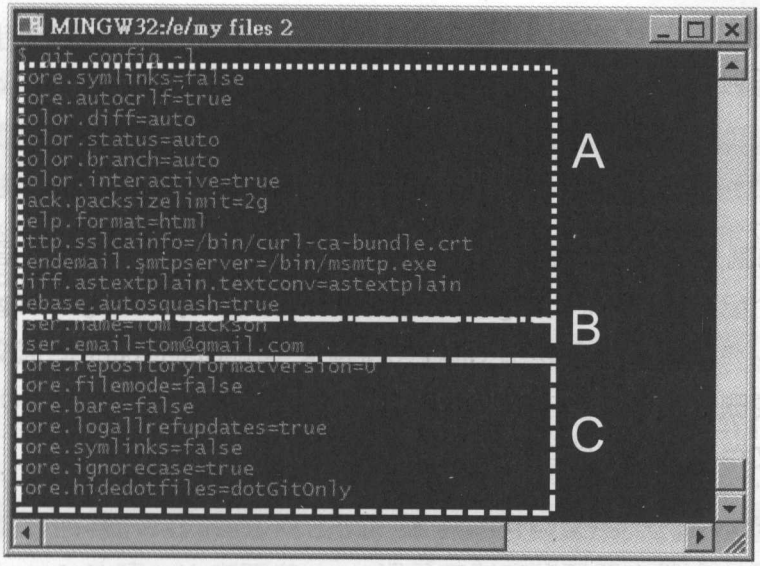


图 2-1　执行“git config –l”指令的显示画面

我们可以在配置文件中添加设置项。例如，在执行“git commit”指令时，必须提供操作者的姓名和 email。上一个单元的做法是使用“--author”选项。这种方式有点麻烦，因为每一次执行“git commit”都要重新输入一次。比较有效率的方法是把操作者的姓名和 email 记录在配置文件中。如此一来，当执行“git commit”指令时，如果没有加入“--author”选项，配置文件中的操作者姓名和 email 就会自动带入。我们可以指定要把操作者的姓名和 email 记录在对应的配置文件中，如果要记录在文档库中的配置文件，可以执行下列指令：

```
git config user.name '操作者姓名'
git config user.email '操作者 email'
```

如果要记录在登录账号的 home directory 中的.gitconfig 文件内，可以执行下列指令：

```
git config --global user.name '操作者姓名'
git config --global user.email '操作者 email'
```

如果要记录在 Git 程序安装文件夹中的 etc\gitconfig 文件内，可以执行下列指令：

```
git config --system user.name '操作者姓名'
git config --system user.email '操作者 email'
```

也就是说，我们是使用“git config”指令的选项来决定要写入到哪一个配置文件内。“--global”表示 home directory 中的.gitconfig 配置文件，“--system”则是指 Git 程序文件夹中的 etc\gitconfig 配置文件。如果要删除配置文件中的设置项，可以使用“--unset”选项。例如要删除文档库配置文件中的操作者姓名，可以执行以下指令：

```
git config --unset user.name
```

如果要删除的是其他配置文件中的设置项，则视情况加入“--global”或是“--system”选项。

补充说明

git 指令的“长”选项和“短”选项

现在我们已经学会了一些 Git 指令的用法，读者会发现有时候 Git 指令的选项是用一个“连字符”开头，像是“-m”，有时候却是用两个“连字符”开头，像是“--author”，为什么呢？其实使用一个“连字符”开头的选项只是简便的形式，我们也可以把它换成完整的形式。例如“git commit -m '说明'”的完整形式是“git commit --message='说明'”（选项后面的“=”可以省略）。又例如“git config –l”的完整形式是“git config --list”。简便形式只是缩短指令的长度，提高输入的方便性，但是并非每一个选项都有简便的形式。我们可以执行“git 指令 --help”调出指令的网页说明文件，里面有完整的指令选项说明。

一般人现在都已经习惯操作图形界面的程序，如果需要输入指令，会觉得有点麻烦。尤其是遇到比较长的指令，如果再加上选项，有时候真的会让人失去耐心。其实我们可以在 Git Bash 程序中，使用

键盘上和下箭头键，调出之前执行过的指令。然后再使用左箭头或是右箭头键，移动到指令需要修改的地方进行修改，最后按下 Enter 键执行即可。另外，为了提高便利性，我们可以定义指令的“别名”（alias），也就是用简短的缩写来表示正规的指令。这样只要输入我们定义的“指令别名”，Git 系统就会自动地把它换成正式的指令。定义指令“别名”的方法如下：

```
git config alias.指令别名 '正式的指令和选项'
```

“指令别名”不可以包含空格，“正式的指令和选项”就是我们输入指令时，在“git”之后输入的那一串文字。如果其中没有空格，就可以省略单引号。另外，如同前面的做法，可以使用“--global”或是“--system”选项，决定要把 alias 记录在对应的配置文件中。我们来看下面一个例子：

```
git config alias.con 'config -l'
```

执行以上指令之后，会在文档库的 config 配置文件中新增一个 alias 设置项，然后就可以使用以下指令显示 Git 的设置，它的结果等同于执行“git config –l”：

```
git con
```

如果要删除这个 alias，同样是使用前面介绍的“--unset”选项：

```
git config --unset alias.con
```

2-2 修改默认的文本编辑器和文件对比程序

执行“git commit”时需要输入说明，比较简短的说明可以直接使用“-m”选项带入。如果说明有好几行，使用文本编辑器会比较方便。读者可以试看看执行“git commit”不加任何选项（假设我们已经在

Git 配置文件中设置好操作者的资料，而且已经执行完“gid add 文件名”），这时候 Git 会启动默认的 vi 编辑器，让我们编辑 commit 说明（提示：如果读者不熟悉 vi 编辑器的操作，可以先输入一个分号“:”，光标会跳到屏幕左下方，接着输入 q，再按下 Enter，就可以离开 vi 编辑器）。如果想要换成自己惯用的文本编辑器，只要在配置文件中加入 core.editor 设置项即可。以下的范例是把默认的编辑器换成 Windows“记事本”程序：

```
git config --global core.editor notepad
```

如果要改用其他文本编辑器，只要把 notepad 换成该程序的路径。如果路径中有空格，可以用单引号或是双引号将它括起来，也可以加上启动程序的选项，例如：

```
git config --global core.editor "'C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession
-noPlugin"
```

设置好之后，下次执行“git commit”指令时，如果不带“-m”选项，就会启动指定的文本编辑器。我们可以在编辑器中输入 commit 说明，输入完毕后，直接用默认的文件名存盘，然后离开。我们输入的 commit 说明就会保存在“.git”子文件夹内的 COMMIT_EDITMSG 文件中，Git 会自动把该文件的内容带入成为 commit 说明。

最后再示范一个高级的配置文件功能，这项功能和文件对比有关，因此我们需要先介绍文件对比的方法。文件对比的目的是要知道我们究竟对文件做了哪些修改。不同类型的文件对比方式都不一样，例如纯文本文件、Word 文件或是图像文件，都需要使用不同的对比方法。Git 系统的主要目的是管理程序代码，因此本身就提供对比程序代码的功能。“git diff”指令就是用来执行文件对比的功能，它可以对比当前文件夹中的文件和文档库中最新的版本，或是已经加入索引的版本（也就是执行过“git add”的文件）之间的差异。假设文档

库中已经存入一个 poem.txt 文件，而且我们又修改了文件夹中的 poem.txt 文件，这时候可以执行“git diff”对比这两个版本之间的差异。以下是“git diff”指令输出的示例：

```
diff --git a/poem.txt b/poem.txt
index d99f9f0..793116e 100644
--- a/poem.txt
+++ b/poem.txt
@@ -1,3 +1,3 @@
-11111
 22222
 33333
+44444
```

第一行的“a/文件名”表示在文档库中或是索引中的文件，“b/文件名”表示在文件夹内的文件。接下来三行是文件的信息，后面以“@@”开头的部分才是文件的差异。“@@”后面的两个数字，也就是“-1, 3”，表示以下数据是 a 文件中第一行开始的后面 3 行，“-”只是表示 a 文件。再后面两个数字，也就是“+1, 3”表示以下数据是 b 文件中第一行开始的后面 3 行，“+”是表示 b 文件。下一行用“-”号开头表示 a 文件变成 b 文件的时候，这一行被删除。用“+”号开头则表示 a 文件变成 b 文件的时候，这一行被加入。没有正负号开头的部分表示没有被修改。

补充说明

1. “git diff”指令显示的信息中可能包含多段以“@@”开头的部分，这种情况表示文件中有多个段落都被修改过，每一个段落的差异都是用相同的方式描述。
2. “git diff”指令可以使用不同的选项来指定要对比哪些版本的文件，后续单元会再详细介绍“git diff”指令的用法。
3. “git diff”指令在显示中文的时候会出现乱码，这时候可以使用“gitk”指令启动图形查看模式。如果我们修改了文件夹中的文件，在 gitk 程序画面左上角会显示一个标示 Local uncommitted changed, not checked in to index 的节点（参见图 2-2）。单击该节点，

（续）

然后单击左下角窗口上方的 Diff，接着在右下方窗口上方单击 Patch，在下方的文件列表中选取要查看的文件，右边的窗口就会显示有修改的部分。

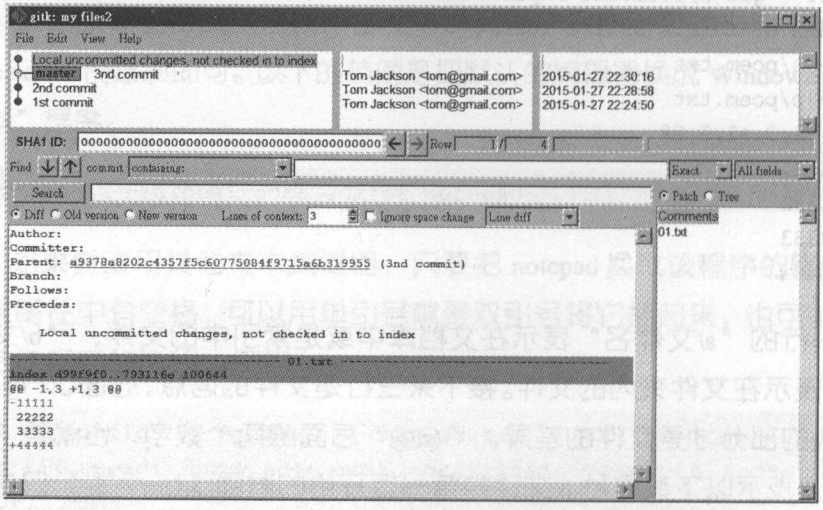


图 2-2　用 gitk 程序找出文件中有修改的部分

了解“git diff”指令的功能之后，或许读者会觉得有点麻烦。如果读者曾经用过 KDiff3 或是 WinMerge 这一类的文件对比程序，一定会觉得 Git 的功能实在太过简单了！没关系，Git 的开发者也知道这个事实。因为 Git 是程序代码版本管理系统，不是专门用来对比文件的。Git 的解决办法就是让程序之间可以相互合作，这样就可以用最完美的方式来完成工作。因此 Git 另外提供一个“git difftool”指令，我们可以在配置文件中指定自己喜欢的文件对比程序，这样 Git 就会把要对比的文件传给这个程序。现在我们就来试试看，请在 Git Bash 程序中执行“git difftool”指令，屏幕画面会显示以下错误信息，表示我们还没有配置文件对比程序：

```
This message is displayed because 'diff.tool' is not configured.
See 'git difftool --tool-help' or 'git help config' for more details.
'git difftool' will now attempt to use one of the following tools:
```

```
kompare emerge vimdiff
No known diff tool is available.
external diff died, stopping at (file name).
```

现在我们就来指定一个文件对比程序，首先必须决定这个程序的名称。假设我们要用 KDiff3，那就把它取名为 kdiff3。另外还要找出它的执行文件的完整路径，然后就可以在 Git Bash 中执行以下指令，完成设置：

```
git config --global diff.tool kdiff3
git config --global difftool.kdiff3.cmd \  ← '\'表示指令还没完成，要延续到下一行
"'D:/Program Files/KDiff3/kdiff3.exe' \"\$LOCAL\" \"\$REMOTE\""
git config --global difftool.prompt false
```

如果要使用不同的文件对比程序，就把其中的 kdiff3 换成自己的程序名称。另外在第三行中的执行文件路径也要按照实际的情况更改。最后一行 git config 指令是关闭提示功能，读者可以把它省略掉，试试看有什么不一样。设置好之后再次执行“git difftool”指令，就会启动 KDiff3 程序，并且显示文件对比的结果，如图 2-3，是不是很棒呢！

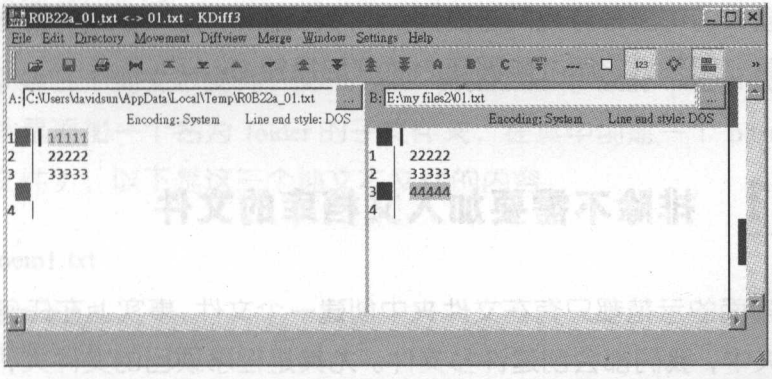


图 2-3　使用“git difftool”指令启动外部文件对比程序



第 3 课

把文件存入 Git 文档库

试用了一些 Git 的功能之后，是不是觉得 Git 很棒呢？Git 还有许多更强大的功能等待我们去发掘！不过先别急，有一些基本概念还是要先建立起来，之后才能够随心所欲地操控 Git。

3-1 排除不需要加入文档库的文件

前面的示范都只有在文件夹中创建一个文件，事实上在任何一个文件夹中，我们都会创建许多文件。尤其是程序项目的文件夹，不但有许多程序文件，还有编译产生的临时文件以及许多子文件夹。需要加入 Git 文档库的只有程序文件和相关的资源文件，编译产生的文件并不需要送进文档库保存。在介绍解决办法之前，我们先解释一下

Git 追踪文件的方式。Git 会将文件和文件夹分成以下三类：

1. 被追踪的（tracked）；
2. 忽略的（ignored）；
3. 不被追踪的（untracked）。

一开始的时候，所有的文件都是 untracked。如果执行“git status”指令，它会列出 untracked 文件列表，这是要提醒我们哪些文件还没有被归类。在正常情况下，文件夹中所有的文件应该被分类成 tracked 或是 ignored。tracked 状态的文件就是已经加入 Git 文档库的文件。要让文件变成 ignored 状态（也就是让 Git 不要检查这个文件），必须先在文件夹中创建一个名为“.gitignore”的文件，然后把要忽略的文件逐一列在这个文件中（一个文件一行）。现在就让我们动手试试看，请按下列步骤操作：

1 步骤 创建一个新的文件夹。

2 步骤 启动 Git Bash 程序，使用 cd 指令切换到这个新的文件夹。

3 步骤 执行“git init”。

4 步骤 创建三个纯文本文件：poem1.txt、poem2.txt 和 folder\poem3.txt（也就是添加一个名为 folder 的子文件夹，在其中创建一个 poem3.txt 文件），以下是这三个纯文本文件的内容：

poem1.txt

```
细草微风岸
危檣独夜舟
```

poem2.txt

```
黄河之水天上来
奔流到海不复回
```

folder\poem3.txt

```
朝辞白帝彩云间
千里江陵一日还
```

5 步骤 回到 Git Bash 程序，执行“git status”指令。屏幕画面会显示以下信息，告诉我们文件夹中有 3 个 untracked 文件和文件夹：

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        poem1.txt
        poem2.txt
        folder/

nothing added to commit but untracked files present (use "git add"
to track)
```

6 步骤 现在我们要把文件 poem1.txt 加入 Git 文档库，请执行以下指令：

```
git add poem1.txt
git commit -m '1st commit'
```

7 步骤 接着创建一个“.gitignore”文件，可以直接在 Git Bash 程序中执行以下指令来创建这个文件：

```
touch .gitignore
```

8 步骤 用文字编辑程序打开“.gitignore”文件，输入以下内容。因为“.gitignore”文件本身就在文件夹内，所以要把它列为 ignored 文件之一。另外指定 folder 文件夹也要 ignored。

```
.gitignore
folder
```

9 步骤 再执行一次“git status”，现在程序会显示只有 poem2.txt 是 untracked。

10 步骤 打开 poem1.txt、poem2.txt 和 folder\poem3.txt 文件，修改内容如下：

poem1.txt

> 细草微风岸
> 危檣独夜舟
> 星垂平野阔 ← 新加入的文字
> 月涌大江流 ← 新加入的文字

poem2.txt

> 黄河之水天上来
> 奔流到海不复回
> 高堂明镜悲白发 ← 新加入的文字
> 朝如青丝暮成雪 ← 新加入的文字

folder\poem3.txt

> 朝辞白帝彩云间
> 千里江陵一日还
> 两岸猿声啼不住 ← 新加入的文字
> 轻舟已过万重山 ← 新加入的文字

11 步骤 再执行一次“git status”指令，现在程序会显示以下信息：

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   poem1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        poem2.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

因为 poem1.txt 是 tracked 状态（也就是属于 Git 文档库中的文件），因此 Git 会追踪它是否被修改（如果要知道修改了哪些内容，可以执行“git diff”或是“gitk”，但是请注意“git diff”无法正确显示中文）。poem2.txt 属于 untracked 状态，所以 Git 一直显示它，但是不会追踪它是否被修改。至于 folder\poem3.txt，因为它属于 ignored 状态，所以 Git 不会在意它是否被修改，也不会显示它的信息。

使用以上范例，我们可以很清楚地了解 Git 如何处理文件夹中的文件和子文件夹。另外关于“.gitignore”文件的用法，我们再做以下补充说明：

1. “.gitignore”文件的影响范围是它所在的文件夹和所有的子文件夹。

2. 每一个文件夹都可以创建自己的“.gitignore”文件，如果它上层的文件夹也有“.gitignore”文件，这个文件夹也会受到它的影响。

3. “.gitignore”文件中可以用“#”字符开头表示注释，文件夹路径是用“/”字符，文件名可以使用通配符“*”。另外还可以使用“!”字符表示排除，例如以下设置表示要忽略所有扩展名是 txt 的文件，但是不包括 note.txt：

```
*.txt
#设置不要忽略 note.txt 文件
!note.txt
```

接下来我们要学习更多 commit 的技巧和管理 Git 文档库的方法。

3-2 控制 Commit

我们已经知道把修改后的文件加入文档库需要 2 个步骤：第一个步骤是用“git add”指令把文件内容加入 Git 的索引。如果文件没有任何修改，执行“git add”就不会有任何效果；第二个步骤是执行“git commit”指令，把 Git 索引中的文件内容加入文档库。但是有时候我们可能在执行“git add”之后反悔，例如加错文件，这时候必须删除已经加入 Git 索引的文件内容。要执行这项操作需要区分以下两种情况：

情况 1：文档库中还没有加入任何文件，也就是执行“git init”之后，还没有执行过“git commit”。

情况 2：文档库中已经有文件，也就是我们已经执行过“git commit”指令。

如果是情况 1，可以使用以下指令删除 Git 索引中特定文件的内容：

```
git rm --cached 文件名
```

如果是情况 2，就必须换成执行以下指令：

```
git reset HEAD 文件名
```

为什么会有这样的差别？这些指令又有什么样的功能？先不急，我们会一一解释。首先要介绍的是“git rm”指令，它是用来删除文档库中的文件或是 Git 索引中的文件，也可以删除文件夹中的文件。如果执行以下指令：

```
git rm 文件名
```

Git 会先执行两项检查：

1. Git 索引中有没有该文件的内容（也就是刚刚有没有执行过“git add

文件名”），如果有，表示这个文件的内容和文档库中的不一样，为了避免遗失数据，Git 会显示提醒信息，然后放弃执行；

2. 文件夹中的文件内容是不是和数据库中的一样，如果不一样，Git 同样会显示提醒信息，然后放弃执行，以免遗失数据。

如果通过以上两项检查，Git 会马上删除文件夹中的文件，然后在索引中记录要从文档库中删除该文件。如果这时候执行“git status”，会显示以下信息，表示将要从文档库中删除该文件：

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      deleted:    poem1.txt
```

最后必须再执行“git commit”指令，才会真正从文档库中删除文件（提示：可以从文档库的历史版本中取回该文件，后续单元会再针对这个主题进行详细的介绍）。

补充说明

1. 请小心使用“git rm”指令，因为它可能会删除文件夹中的文件。
2. 如果确定要从文件夹和 Git 文档库中删除文件，除了使用“git rm”指令之外，还有一个更方便的做法。我们可以先在文件夹中删除文件，然后执行“git add -A .”（“-A”是短选项，也可以换成长选项“--all”）。Git 会把文件夹中已经不存在的文件记录在索引中，接着再执行“git commit”，就会从文档库中删除这些文件，但是从 commit 的历史版本中还是可以获取这些文件。

如果执行“git rm”时加入“--cached”选项如下：

```
git rm --cached 文件名
```

表示从此以后不需要在文档库中更新这个文件，也就是这个文件将从 tracked 状态转变成 untracked 状态，Git 索引中这个文件的内容也会被

删除。但是，Git 不会从文件夹中删除这个文件（这就是加上“--cached”选项最大的差别）。了解“git rm”指令的原理之后，让我们回到前面讨论的情况 1，因为文档库中当前还没有任何文件，所以我们执行“git rm --cached 文件名”之后，该文件就是从 Git 索引中删除而已，不会对文档库造成任何影响。可是，如果文档库中已经有该文件，执行“git rm --cached 文件名”之后，该文件就会回到 untracked 状态，但是原来我们只是想要将它从索引中删除而已，并不想要改变它的 tracked 状态，这样的结果和我们想要达到的结果显然不同。

现在剩下的问题是，为什么情况 1 不能够使用“git reset HEAD 文件名”这个指令？原因在于 HEAD 这个关键词，要解释它需要进一步说明 commit 节点之间的关系。

3-3 查看 Commit 节点

我们已经知道如何使用“gitk”指令启动图形查看模式。在这种操作模式下，左上角的窗口会显示文档库中 commit 节点的历史演进图。用鼠标单击其中的节点，窗口下方的“SHA1 ID”字段就会显示该节点的标识符，如图 3-1 所示。在文档库中，每一个 commit 节点之间都有明确的先后次序关系，我们用图 3-2 来说明。

补充说明

除了使用 gitk 程序查看 commit 节点的信息以外，也可以使用“git log”指令达到相同的目的。如果加上“--graph”选项，会用文本模式排列出 commit 节点的演进图，另外可以再搭配“--oneline”选项，用最简便的方式显示，读者可以试试看。

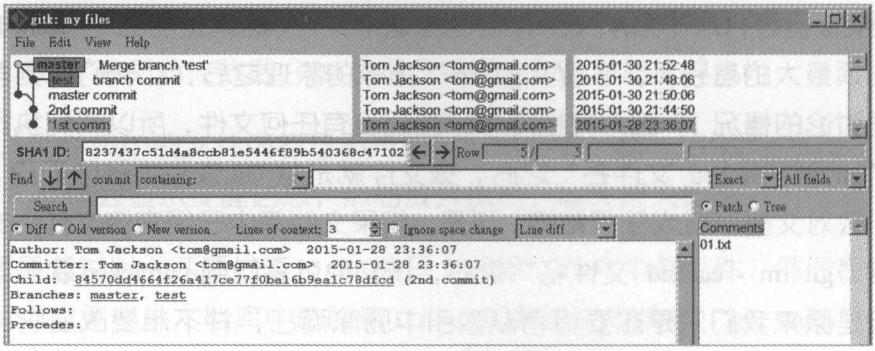


图 3-1　使用 gitk 程序查看 commit 节点的标识符

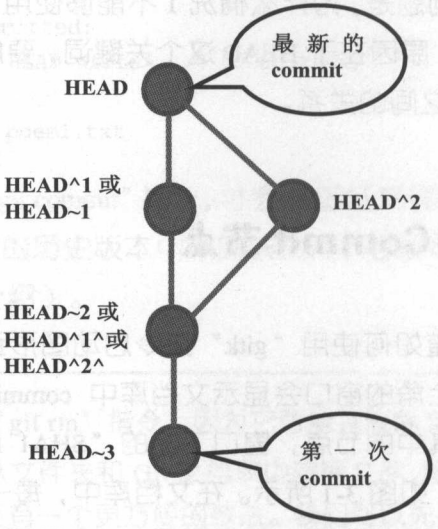


图 3-2　文档库中 commit 节点的演进示意图

图 3-2 最上面的节点代表最新的 commit，最下面的节点是第 1 次 commit。Git 系统会赋予每一个 commit 节点一个独一无二的标识符。除了节点的标识符之外，Git 还提供一个叫作 HEAD 的节点标签，这个节点标签永远代表最新的 commit。例如我们可以执行以下指令，显示最新 commit 的详细数据：

```
git show HEAD
```

由于 commit 节点之间有先后次序的关系，我们可以从某一个节点开始，找到比它更早的节点。要指定某一个节点之前的节点可以使用以下格式：

```
节点标签或是标识符^数字
节点标签或是标识符~数字
```

其中的数字是当该节点有一个以上的父节点时（如图 3-2 的 HEAD 节点，这是因为文档库中有 branch 的情况，后续单元会再详细介绍 branch 的功能和使用方法），用来指定是哪一个父节点。如果只有一个父节点，就可以省略数字。我们可以重复使用“^”字符表示父节点的父节点，依此类推，如图 3-2 所示。除了“^”字符之外，还有“~”字符，它可以用来表示哪一层的父节点，例如“HEAD~1”表示 HEAD 第一层的节点，“HEAD~2”表示 HEAD 第二层的父节点，依此类推，请参考图 3-2 的范例。

补充说明

1. commit 节点的标识符是一组很长的 16 进制数字，在指定 commit 节点标识符的时候，不需要将它完整列出，一般只要使用最前面 4 个数字即可，Git 会自动找出对应的节点。如果找到超过一个以上的节点，Git 会显示错误信息，这时候换用长一点的数字就可以解决。
2. HEAD 这个节点标签还可以简写成“@”，也就是说“HEAD~1”可以写成“@~1”，“HEAD^2”可以写成“@^2”，依此类推。
3. “git show”指令后面也可以接文件名，这时候会显示该文件最新版本的修改情况，也就是比较文件最新版本和前一个版本的差异。如果使用格式“git show commit 节点:文件名”，表示要显示指定的 commit 节点中特定文件的内容。

除了 Git 内设的 HEAD 节点标签之外，我们也可以自己帮 commit 节点“贴上”自定义的标签，指令格式如下：

```
git tag 自定义的标签名称 commit 节点标识符或是标签
```

将 commit 节点贴上标签之后，就可以像 HEAD 一样使用我们自定义的节点标签。如果要删除自定义的节点标签，可以使用以下指令：

```
git tag -d commit 节点标签
```

了解 HEAD 标签的功能之后，现在回到前面讨论的情况 2，也就是使用“git reset HEAD 文件名”，将文件内容从 Git 索引中删除。为什么这个指令不能用于情况 1？因为当文档库中没有任何文件时，表示这个文档库没有执行过任何 commit，因此 HEAD 是空的。这时候如果使用 HEAD 标签，会出现错误信息。最后我们再补充一些“git reset”指令的用法，它可以对文档库进行“消磁”的操作，也就是让文档库恢复到某一个节点的状态，它的指令格式如下：

```
git reset 选项 commit 节点标识符或是标签
```

其中的选项如果是“--soft”，表示只有文档库里面的数据会改变，Git 索引和文件夹中的文件都不会受到影响。如果选项是“--mixed”（这是默认的选项），表示 Git 索引也会恢复到指定节点的状态，但是文件夹中的文件仍然不会受到影响。如果选项是“--hard”，则文档库、Git 索引和文件夹中的文件，都会恢复到指定节点的状态。



第 4 课

比较文件的差异和从 Git 文档库取回文件

到当前为止，我们介绍的 Git 功能主要是把文件存入文档库。这些存储在文档库中的文件就是我们的“历史备份”，等到需要它们的时候，再从文档库中取出。有些时候，我们也想知道文件修改的过程，也就是比较不同版本之间的差异。Git 提供了非常完整的文件对比方式，这样就可以让我们对比任何版本的文件。不管是在文件夹内，或是在 Git 索引中，或是已经存储在文档库的历史版本，都可以互相对比。Git 的文件对比指令，其实就是前面单元已经使用过的“git diff”和“git difftool”，当时的重点是在介绍配置文件的用法，因此没有讲解完整的功能。其实，这两个指令可以搭配不同的选项和参数，来指定要对比的文件版本。不过它的用法有点复杂，为了帮助读者了解，下面把不同选项和参数的搭配整理如图 4-1。

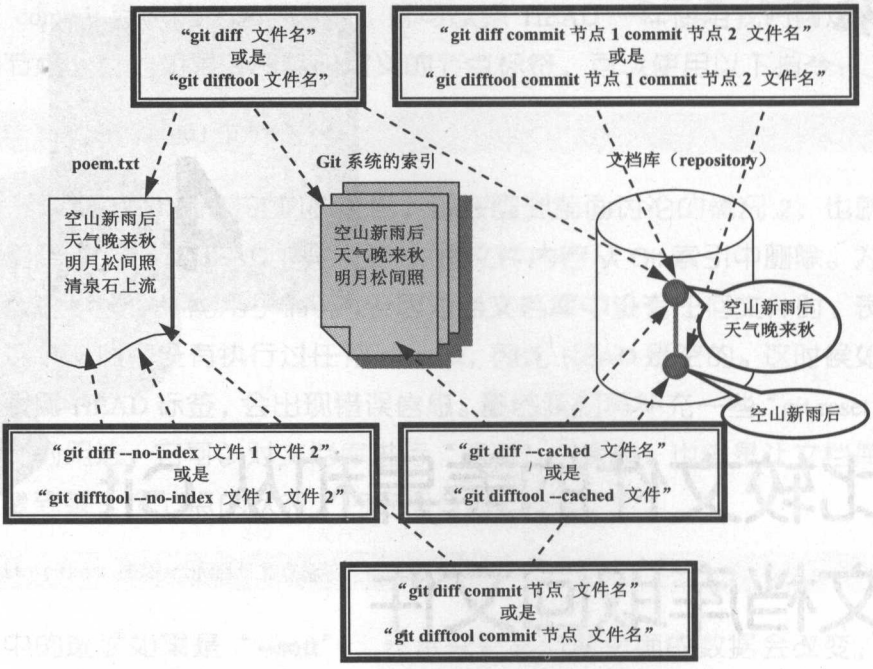


图 4-1　"git diff" 和 "git difftool" 指令的选项和参数搭配方式及其功能

上图中有一个名称叫作 poem.txt 的文件，它的内容总共有 4 个版本。最早的 2 个版本已经使用 "git commit" 指令存入文档库（图 4-1 最右边的部分）。另外一个比较新的版本是记录在 Git 索引中（使用 "git add poem.txt" 指令）。但是在执行过 "git add" 指令之后，我们又修改了它的内容，也就是图 4-1 最左边的版本。图 4-1 中矩形框内部的指令，可以让我们对比这些不同的文件版本。举例来说，如果要对比文档库中任何两个 commit 版本的差异，可以使用图 4-1 右上方矩形框中的指令格式。例如执行以下指令之后，会得到图 4-2 的结果（假设我们已经按照单元 2 介绍的方法设置好 difftool 的功能）。

```
git difftool HEAD HEAD^ poem.txt
```

补充说明

如果要对比全部的文件，可以将文件名换成 "."，也就是 "git difftool HEAD HEAD^ ."，或是直接省略文件名也可以。

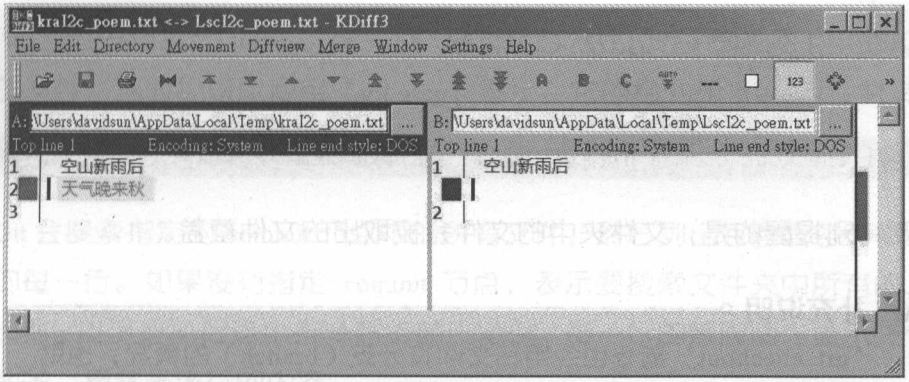


图 4-2　执行“git difftool HEAD HEAD^ poem.txt”之后显示的画面

另外要特别说明的是，如果执行“git diff”或是“git difftool”指令时不带任何选项，表示要确认在执行“git add”或是“git commit”之后文件有没有再被修改。Git 会先找索引中有没有新的版本，如果有，就拿这个版本和文件夹中的文件对比，看看是不是相同（提示：Git 索引中的内容必须执行“git commit”之后才会加入文档库）。如果 Git 索引中没有新的文件版本，就用文档库中最新的版本和文件夹中的文件对比。如果执行“git diff”之后没有显示任何结果，就表示文件没有再被修改。另外也请读者留意“--no-index”这个选项（图 4-1 左下角的指令格式），它可以用来对比文件夹中任何两个文件。但是，如果加上这个选项，就算是执行“git difftool”指令，也不会启动我们设置的文件对比程序。也就是说，这个选项一律使用 Git 内部的对比程序。不过，这个限制并不会造成实质的影响，因为既然文件是在文件夹内，我们可以自己启动外部的文件对比程序来进行对比。

4–1　从 Git 文档库中取出文件

把文件存入文档库与把文件取出文档库相比，从文档库取出文件要简单得多，因为它只需要一个指令，就是“git checkout”。我们可

以使用下列指令格式，从文档库的任何一个 commit 节点取出指定的文件：

```
git checkout commit 节点标识符或标签 文件 1 文件 2 …
```

要特别提醒的是，文件夹中的文件会被取出的文件覆盖。

补充说明

“git checkout”指令也会用在文件有分支（branch）的情况，也就是文件在同一个时间，出现一个以上修改的版本（通常发生在多个人同时修改一个文件的情况），后续我们会再详细介绍这种情况。

执行“git checkout”指令时，有一种情况要特别留意，如果取出的文件和当前文档库中最新 commit 的文件内容不同，这个取出的文件内容会自动记录在 Git 索引中。如果之后我们又执行“git commit”指令，这个取出的文件内容就会存入文档库中成为新的版本。如果要避免这种情况发生，可以在执行“git checkout”指令之后，立刻执行“git reset HEAD”，清除 Git 索引。

执行“git checkout”指令时，如果指定的 commit 节点没有包含指定的文件，Git 会自动从更旧的 commit 节点寻找该文件，再从中取出最新的版本。如果都没有找到该文件，就会显示以下信息：

```
error: pathspec 'poem2.txt' did not match any file(s) known to git.
```

如果执行“git checkout”指令时没有指定 commit 节点，也就是使用以下格式：

```
git checkout 文件 1 文件 2 …
```

Git 会先找索引中有没有该文件，如果有就把它取出；如果没有，就从最新的 commit 节点开始，按照时间顺序往前寻找，然后取出第一个找到的文件版本，每一个文件都用同样的方式处理。如果要取出文档库中全部文件的最新版本，可以执行“git checkout .”。

另外还有一个常见的情况，就是我们想要找出包含某一个字符串的文件。这时候可以使用“git grep”指令，它的格式如下：

```
git grep '要找的字符串' commit 节点
```

Git 会搜索指定的 commit 节点中的所有文件，然后列出包含该字符串的每一行。如果没有指定 commit 节点，表示要搜索文件夹中所有的文件。以下是执行这个指令后的输出范例，每一行的最前面先显示文件名，接着是该行的内容：

```
GameResultFragment.java: frag.mEdtCountSet = (EditText) getActivity()
MainFragment.java:     public EditText mEdtCountSet,
```

这个指令默认会区分英文大小写。如果不要区分英文大小写，可以加上“-i”选项。如果只要列出包含该字符串的文件，可以加上“-l”选项。如果还想知道每一个文件中有几行含有该字符串，可以换成使用“-c”选项。例如以下是加上“-c”选项的结果，文件名后面的数字就是行数：

```
GameResultFragment.java:1
GameResultFragment2.java:1
MainFragment.java:12
```

如果要找的字符串不止一个，可以使用“-e”选项分别指定这些字符串：

```
git grep -e '要找的字符串 1' -e '要找的字符串 2' commit 节点
```

这些字符串默认是以 or 的方式结合，也就是说，只要出现其中一个，就算符合条件。如果要限制所有的字符串都必须出现才算数，可以用“--and”选项把它们结合起来如下：

```
git grep -e '要找的字符串 1' --and -e '要找的字符串 2' commit 节点
```

and 和 or 的结合还可以混用，甚至加上括号如下：

```
git grep -e '要找的字符串 1' --and \( -e '要找的字符串 2' -e '要找的字符串 3'
\) commit 节点
```

以上指令的功能是要显示含有字符串 1，而且含有字符串 2 或是字符串 3 的所有行。

如果想要知道文件中每一行最后是由谁修改，可以执行“git blame 文件名”这个指令。如果文件太长，还可以使用“-L”选项指定从哪一行开始到哪一行结束，请参考以下格式范例：

```
git blame -L 起始行，结束行 文件名    ← 显示从起始行到结束行的数据
git blame -L 起始行，文件名           ← 显示从起始行到文件最后一行的数据
git blame -L,结束行，文件名           ← 显示从文件开始到结束行的数据
```

4-2 使用“git mv”指令改变文件或是文件夹的名称

如果我们更改文件夹中的文件名或是子文件夹名称，Git 文档库也要进行同样的修改。我们可以使用前面单元学过的指令来完成这项工作，步骤如下：

1 步骤 使用 Windows 文件资源管理器更改文件名或是文件夹名称。

2 步骤 回到 Git Bash 程序，执行“git add -A 新文件名”，把更改文件名的文件加入 Git 索引，同时在 Git 索引中记录被删除的文件。如果更改的是文件夹名称，也是用同样的方式处理。

3 步骤 执行“git status”指令，屏幕画面会显示以下信息。Git 很聪明，它按照文件内容，判断新增的文件和被删除的文件是同一个文件，只不过名称不一样。因此会用变更名称的方式存入文档库，以避免重复存储相同的文件内容。

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    原来的文件名 ->新文件名
```

4 步骤 执行“git commit”指令，将索引的内容写入文档库。

虽然以上步骤可以达到更改文件名的目的，也能够更新 Git 文档库。可是需要多个操作步骤。其实只要使用“git mv”指令，就可以做到相同的功能，它的用法如下：

```
git mv 原来的文件名 新文件名
```

这个指令会更改文件夹中的文件名，然后把它记录在 Git 索引。接着只要执行“git commit”指令，就可以更新文档库。如果要更改文件夹的名称，也是用同样的方式处理。

4-3 暂存当前文件夹的文件状态

这项功能可以让我们把当前文件夹中的“文件修改状态”暂存起来，以便稍后将它们恢复。例如当程序修改到一半的时候，突然收到一个临时性的工作，需要先修改其他地方。但是当前的工作只进行到一半，不适合将它存入文档库。这时候就可以使用暂存的功能，先将当前的文件状态记录下来，然后开始处理临时的工作。等到临时工作处理完毕之后，再将文件夹中的文件恢复到原来的状态。

要暂存文件夹中的文件状态可以使用“git stash save”指令，这个指令会执行下列两项工作：

1. 存储文件夹中被 Git 追踪的文件和文档库中最新文件版本的差异；
2. 把文件夹中被 Git 追踪的文件，还原成文档库中最新的文件版本。

执行“git stash save”指令之后，我们可以接着执行“git stash list”，它会显示如下的信息：

```
stash@{0}: WIP on (no branch): 01325b4 3rd commit  ← 执行“git stash save”时对比的基础文件版本
```

其中的 WIP 是 work in progress 的缩写，用来说明当前进行中的工作内容。信息最后会显示当前文档库中最新的 commit 节点标识符和说明，它也是当前文件夹中文件的版本。

执行完“git stash save”指令之后就可以开始着手另一项工作，如修改当前文件夹中的程序文件，或是从文档库中取出其他文件版本来修改。等处理完毕之后，将它存入文档库。接下来必须让文件夹中的文件，恢复成执行“git stash save”指令时的状态，这需要执行下列两个步骤：

1. 先从文档库中取出执行“git stash save”指令时的对比文件版本，也就是执行“git stash list”所显示的 commit 版本，我们可以使用“git checkout commit 标识符 .”完成这个步骤；

2. 执行“git stash pop”或是“git stash apply”取出暂存的文件，并且合并到当前文件夹中的文件。

完成以上两个步骤之后，文件夹中的文件就会恢复到执行“git stash save”指令时的状态。如果执行“git stash pop”或是“git stash apply”成功，暂存的文件状态会被清除，这时候再执行“git stash list”就不会显示任何信息。如果执行过程中出现错误，Git 会保留临时文件，以免丢失数据。

4-4 清理 Git 文档库

Git 文档库经过一段时间的存取和更改之后，里面的数据可能会变

得比较零散，而且有些数据已经不会再使用。虽然这种情况不会影响 Git 的正常运行，但是会让效率变低，而且还会浪费磁盘空间。为了改善这个问题，我们需要清理 Git 文档库。其实 Git 在执行某些指令之后，也会自动清理文档库，例如，后续单元会介绍的“git rebase”指令。执行这个指令可能造成文档库中大量的数据变动，因而可能让数据变得比较零散，这时候 Git 会自动清理文档库。除了让 Git 自动清理之外，有时候我们也要自己执行清理文档库的操作，以确保它的运行效率。

清理 Git 文档库其实很容易，只需要执行“git gc”指令即可（gc 是 garbage collection 的缩写）。它有一些搭配的选项如下：

1. “--aggressive”

 Git 在默认情况下，会用比较快速的方式检查文档库，并完成清理。如果加入这个选项，Git 会用比较仔细的方式检查以及清理，但是需要比较久的时间。这个选项只需要“偶尔”使用即可，太常使用会浪费时间，不会有明显的帮助。

2. “--auto”

 如果加入这个选项，Git 会先判断文档库是否需要清理。如果情况还算良好，就不会执行清理的操作。

3. “--no-prune”

 这个选项是要求 Git 不要“清除”文档库中不会用到的数据，只要“整理”它们即可。

从第一个单元到现在，我们已经学会如何创建 Git 文档库、将文件存入文档库、查看和管理文档库，以及从文档库取出文件。下一个单元我们会用一个实际的程序项目为例，实践演练这些操作，体验一下 Git 强大的程序代码管理能力。



第 5 课

Git 程序项目管理的实践演练

到当前为止，我们已经了解 Git 如何工作，并且学会了一些基本的操作技巧。这个单元我们就来实践演练一下，看看如何把它们运用到实际的程序项目中。我们将以 Android App 项目开发为例，看看如何使用 Git 管理程序代码。虽然我们是用 Android App 项目来示范，但是同样的技巧可以适用其他的程序项目。

当我们使用 Android Studio 开发工具创建一个新的 App 项目时，会在 App 项目文件夹中产生许多文件和子文件夹，其中包括项目配置文件、程序文件、链接库文件、各种项目资源文件，以及编译程序项目所产生的暂存文件。需要由 Git 管理的文件只有项目配置文件、程序文件、链接库文件和项目资源文件，编译过程中产生的暂存盘并不需要加入 Git 文档库。我们可以使用第 3 单元中介绍的“.gitignore”文件

来排除这些暂存盘。不过 Android Studio 很贴心，它在创建 App 项目的时候，会自动帮我们生成默认的“.gitignore”文件，省去了我们自己准备这个文件的麻烦。因此创建新的 App 项目之后，我们可以立刻把它存入 Git 文档库。假设现在已经创建好一个新的 App 项目，接下来就可以启动 Git Bash 程序，使用“cd”指令切换到 App 项目的文件夹，然后执行以下指令：

```
git init      ← 创建 Git 文档库，让 Git 开始管理这个文件夹中的文件
git status    ← 显示当前文件夹中的文件状态
```

屏幕画面会显示以下信息，表示有哪些文件和子文件夹没有被加入到 Git 文档库（也就是还没有被 Git 追踪的文件和子文件夹）：

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        .idea/
        AndroidApp.iml
        …(所有未被追踪的文件和子文件夹的列表)

nothing added to commit but untracked files present (use "git add" to
track)
```

接下来执行以下指令，把所有文件和子文件夹（除了被“.gitignore”文件排除的以外）存入文档库：

```
git add .
git commit -m 'Initial app project.'
```

现在我们可以执行“gitk”指令，使用图形查看模式观察文档库的内容。左上角的窗口中会有一个 commit 节点（如图 5-1），它就是刚刚执行“git commit”指令的结果。既然我们已经把程序项目加入

Git 文档库，现在就来测试一下，看看 Git 是不是真的帮我们把文件存储起来。先关闭 Android Studio，然后把项目文件夹中的文件全部删除（请注意，不要删除“.git”这个子文件夹，因为它是 Git 存储数据的地方）。接着再使用“git checkout HEAD .”指令，从 Git 文档库中取出所有的文件。最后重新启动 Android Studio，打开这个 App 项目，结果应该和原来的状态一模一样。

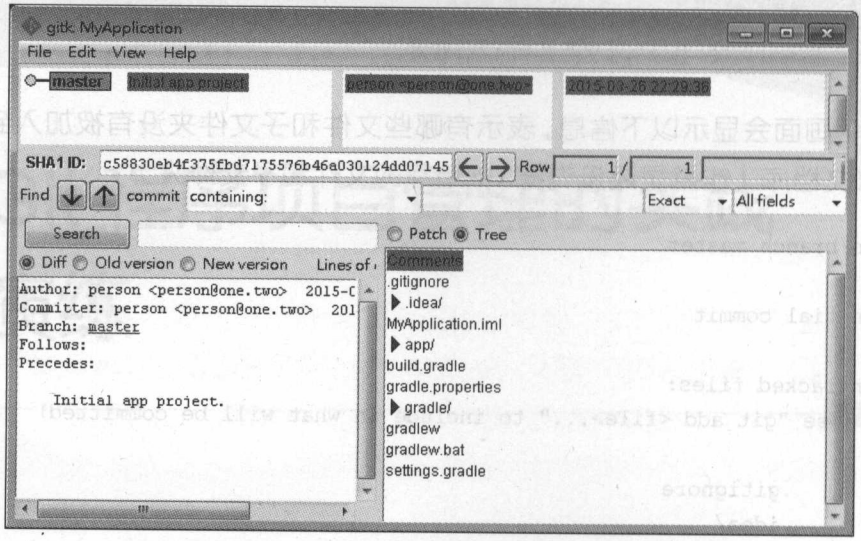


图 5-1　使用 gitk 图形查看模式查看新加入的程序项目

接下来我们要帮 App 加入新功能，假设已经加入一个新的按钮，并且完成相关的程序代码。现在要将这个修改后的版本加入 Git 文档库。先切换到 Git Bash 程序，然后执行以下指令：

```
git add .
git status
```

屏幕画面会显示以下信息，告诉我们有哪些文件被修改过，并且已经准备好要存入文档库：

```
On branch master
Changes to be committed:
```

```
  (use "git reset HEAD <file>..." to unstage)

        modified:   .idea/misc.xml
        modified:   app/src/main/java/com/androidapp/MainActivity.java
        modified:   app/src/main/res/layout/activity_main.xml
        modified:   app/src/main/res/values/strings.xml
```

最后执行以下指令，把修改后的内容存入文档库：

```
git commit -m 'Add a button.'
```

以上的操作只有更改了文件内容，并没有在程序项目中加入了新的文件，接下来试试看添加了新文件的情况。假设我们在 App 中创建了一个新的 GameFragment 类，包括它的程序文件和界面布局文件。然后切换到 Git Bash 程序，接着同样执行“git add .”和“git status”指令，屏幕画面会显示以下信息，告诉我们有哪些新文件，以及有哪些文件被修改了：

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   app/src/main/java/com/androidapp/GameFragment.java
        new file:   app/src/main/res/layout/fragment_game.xml
        modified:   app/src/main/res/values/strings.xml
```

最后执行“git commit”指令，把它们存入文档库。我们可以再使用“gitk”指令查看当前文档库的状态。

现在假设我们想要反悔，就要从项目中删除这个新加入的 GameFragment。这时候除了从文件夹删除它的相关文件以外，还要从 Git 文档库删除这些文件的追踪状态。虽然前面单元学过的“git rm”指令可以达到这个目的，但是还有一个更简便的方法，请按照以下步骤操作：

1 步骤 从 App 项目的文件夹中删除不想要的文件。

2 步骤　切换到 Git Bash 程序，执行“git add -A .”。

补充说明

执行“git add .”指令时如果没有加上“-A”选项，Git 则不会检查文件夹中是否有文件被删除。但是如果加上“-A”选项，Git 就会检查文件夹中是否有文件被删除。这些被删除的文件会在 Git 索引中标记，当执行 commit 指令的时候，被标记删除的文件也会从新的 commit 节点中删除。

3 步骤　执行“git status”指令，屏幕画面会显示 Git 索引中已经标记好要删除的文件。

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    app/src/main/java/com/androidapp/GameFragment.java
        deleted:    app/src/main/res/layout/fragment_game.xml
```

4 步骤　最后再使用“git commit”指令，从新的 commit 节点中删除这些文件。

以上操作只是取消这些文件的追踪状态，其实这些文件的内容还是留在文档库的历史版本中，我们可以使用前面单元学过的“git checkout”指令取回这些文件。如果想要了解某一个文件的修改记录，也可以使用前面学过的“git difftool”指令进行文件的对比，例如：

```
git difftool HEAD~1 HEAD~2 app/src/main/res/values/strings.xml
```

再者，如果想要让文档库恢复到以前某一个 commit 节点的状态，也可以借助“git reset”指令。总之，我们可以熟练使用前面单元介绍的指令和操作技巧，让 Git 帮我们保存程序项目开发的历史记录，并且可以随时查看，或是取回任何版本的文件。



第 6 课

获取 Git 文档库统计数据和绘制统计图表

随着程序项目的开发，Git 文档库中的程序代码会不断地增长。文件数目也会持续增加，甚至参与开发程序项目的人员也会越来越多。为了了解程序项目的状态，我们需要知道当前 Git 文档库中究竟有多少个文件以及程序代码有多少行。甚至想要进一步了解程序项目更新的频率（也就是执行 commit 的频率）和修改的幅度（也就是程序代码到底修改了多少行），或是想要知道不同程序开发人员 commit 程序代码的情况。诸如此类的数据，可以使用 Git 指令，再搭配适当的工具程序来获取。另外，也有开放源码的程序项目可以帮我们计算 Git 文档库的统计数据，并且绘制成图表。这个单元我们就来介绍如何完成这些工作。

补充说明

虽然到当前为止，我们是以单人开发程序项目为例来介绍 Git 指令的操作技巧，但是，其实团队开发模式才更需要 Git 的协助。在这种情况下，Git 文档库中会有不同开发人员的 commit 节点。本书后续单元会再详细介绍相关的指令和操作方式。

6-1 获取 Git 文档库的统计数据

Git 文档库的统计数据包括参与开发程序项目的人员、commit 次数以及程序代码行数，另外我们也可以设置数据统计的时间日期。以下是相关指令的用法：

1. “git log” 指令

这个指令会按照时间顺序，从最近一次的 commit 节点开始，往前列出每一次 commit 的数据，包括标识符、执行人、日期和时间、以及说明。其实如果只要查看 commit 节点的数据，使用 gitk 图形查看模式会更方便。但是，“git log” 指令的优点是可以搭配不同的选项来筛选出特定的 commit。例如加上 “--author” 选项可以指定只列出特定人的 commit：

```
git log --author='A'
```

以上指令只会显示以大写字母 A 开头的人名所执行的 commit。如果只要显示更改过某些文件的 commit，可以使用以下指令（文件的更改包括：加入、删除以及修改内容）：

```
git log 文件名1 文件名2 …
```

另外还可以使用 “--after”（也可以换成 “--since”）或是 “--before”（也可以换成 “--until”）来设置要显示的 commit 时间日期，例如：

```
git log --after='2015-02-18 9:30' --before='2015-02-18 21:50'
```

以上指令只会显示指定期间的 commit。如果加上“--stat”、“--numstat”或是“--shortstat”，还可以显示每一个 commit 变更的情况，例如以下信息是执行“git log –shortstat”的结果：

```
commit 802624d21bfc516e96bc87e6b5fe0af63571486e
Author: AAA <AAA@AAA.AAA>
Date:   Wed Feb 18 21:28:17 2015 +0800

    Add button 1

 2 files changed, 13 insertions(+), 1 deletion(-)

commit 1c7633bab0f4422e631c4853902b3379f68efcd3
Author: AAA <AAA@AAA.AAA>
Date:   Wed Feb 18 21:25:59 2015 +0800

    Initial project

 35 files changed, 656 insertions(+)
```

除了原来 commit 的信息以外，在每一个 commit 最后还会显示有多少文件被修改，以及增加了几行程序代码或删除了几行程序代码（粗体字标记的部分）。如果加上“--oneline”选项，还可以用更精简的方式显示如下：

```
802624d Add button 1
 2 files changed, 13 insertions(+), 1 deletion(-)
1c7633b Initial project
 35 files changed, 656 insertions(+)
```

如果想要分别汇总新增的程序代码和删除的程序代码，可以搭配其他的工具程序。也就是使用命令行模式的管道（pipe）功能，将 Git 指令的输出传给另一个工具程序处理，最后筛选出我们想要的结果。以下是显示新增程序代码和删除程序代码总和的范例：

```
git log --pretty=tformat: --numstat | gawk '{ add += $1 ; subs +
= $2 ; loc += $1 - $2 } END { printf "added lines: %s removed lines : %s
total lines: %s\n",add,subs,loc }'
```

执行上述指令之后会显示如下的结果：

```
added lines: 811 removed lines : 1 total lines: 810
```

补充说明

如果 Git 指令显示的信息超过屏幕高度，屏幕画面会自动暂停。这时候可以使用以下按键进行后续操作：

1. 按键 j 表示显示下一行；
2. 空格键表示显示下一页；
3. 按键 h 表示显示操作说明；
4. 按键 q 表示结束。

2. “git shortlog” 指令

这个指令会按照名字的字母顺序，列出每一个人执行 commit 的次数和说明，如以下范例，其中的 AAA、BBB 和 CCC 是人名，姓名后面括号内的数字就是 commit 次数：

```
AAA (3):
     Initial project
     Add button 1
     Add GameFragment

BBB (1):
     Add button 2

CCC (1):
     Add OK button in GameFragment
```

我们也可以按照 commit 次数，从高至低按序排列，只要加上“--numbered”选项（或是短选项“-n”）即可。如果不需要显示 commit 说明，可以再加上“--summary”选项（或是短选项“-s”），

就会得到以下结果：

```
  3 AAA
  1 BBB
  1 CCC
```

3. “git ls-files” 指令

这个指令会列出当前 Git 文档库中的文件列表，我们可以搭配 xargs 和 wc 这两个工具程序，来计算文件中的程序代码行数，例如执行以下指令：

```
git ls-files | xargs wc -l
```

会显示下列结果：

```
  21 app/src/main/AndroidManifest.xml
 109 app/src/main/java/com/androidapp/GameFragment.java
  39 app/src/main/java/com/androidapp/MainActivity.java
```

最前面的数字是文件中的行数。如果要计算行数的总和，可以将指令修改如下：

```
git ls-files | xargs cat | wc -l
```

学会如何从 Git 文档库中获取统计数据之后，接下来要介绍如何使用第三方程序，画出 Git 文档库的统计图表。

6-2 使用 GitStats 绘制统计图表

GitStats 是一个开放源码的程序项目，它是用 Python 语言开发的。首先它会使用前面介绍的 Git 指令，从文档库中获取统计数据。然后通过 gnuplot 绘图软件包，将数据绘制成统计图表。使用 GitStats 之前必须先安装相关的软件，请读者按照以下步骤完成安装：

1 步骤 首先是安装 Python 程序设计语言。Python 是一个广受好评的解释性程序设计语言，许多人因为喜欢它，所以用它开发各种各样的软件包，将它带入各种应用领域，例如网站架设、绘制图表……等。GitStats 就是使用它的图表绘制功能来显示 Git 文档库的统计数据。请读者打开网页浏览器，连接到以下网址，下载最新的 Python 2.X 版进行安装。在设置安装选项时，记得把 Python 执行文件的路径加入 Windows 的 Path 环境变量（如图 6-1），这样才能够在 GitStats 的文件夹中执行 Python 程序。

https://www.python.org/downloads/

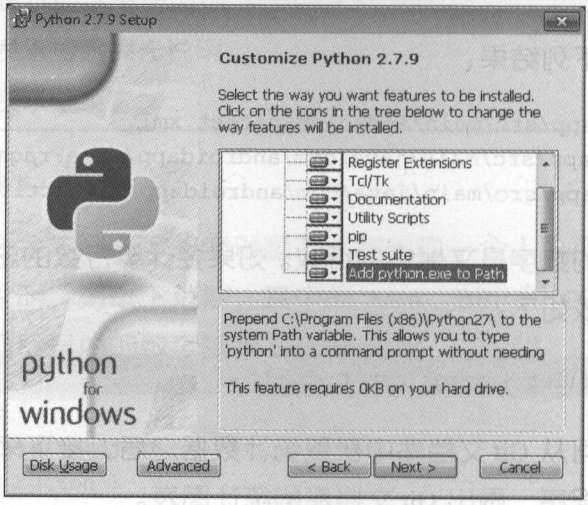


图 6-1　安装 Python 时将 Python 执行文件的路径加入 Path 环境变量

补充说明

为什么不安装 Python 3.X？Python 2.X 和 3.X 在语法上有许多差异，有些 Python 程序是用 Python 2 的语法编写的，这些程序如果使用 Python 3.X 执行，会出现语法错误，GitStats 就是其中之一。

2 步骤 安装 gnuplot 软件包，它是一个跨平台的图表绘制软件包。请读者打开以下网址，下载最新版本进行安装。如果想要多了解一些

gnuplot 的信息，可以参考官方网站 http://www.gnuplot.info/。

http://sourceforge.net/projects/gnuplot/files/gnuplot/

3 步骤 接下来也要像 Python 一样，把 gnuplot 的执行文件路径加入 Windows 的 Path 环境变量，这样 GitStats 执行的时候才能够找到它。请选择 Windows 的开始→控制台，打开控制面板中的“系统”选项，在左边的选项列表中找到“高级系统设置”。单击它之后，在出现的对话框下方单击“环境变量”按钮。在下一个对话框的“系统变量”列表中单击 Path，然后单击下方的“编辑”按钮，在“变量值”字段的最后输入“;(安装 gnuplot 的路径)\gnuplot\bin”，完后之后单击确定按钮。

补充说明

Git 软件的执行文件也要包含在 Path 环境变量中，否则执行 GitStats 程序时会出现错误。将 gnuplot 执行文件的路径加入 Path 环境变量时，可以顺便检查一下。如果没有，请按照同样的方式处理。

4 步骤 最后就是安装 GitStats 程序，它其实是一个公开在网络上的开放源码程序项目。我们可以连接到以下网址，单击网页右下方的 Download ZIP 按钮。下载完毕之后，将压缩文件解压缩到一个新的文件夹。然后找到里面一个名为 gitstats 的文件，将它的扩展名加上 py 成为 gitstats.py。

https://github.com/hoxu/gitstats

完成以上步骤之后，就可以使用 GitStats 程序来产生 Git 文档库的统计图表。GitStats 程序会创建一组网页文件，使用网页浏览器，打开其中的 index.html，就可以看到这些图表。执行 GitStats 程序的步骤如下：

1 步骤 打开 Windows 命令行窗口。

2 步骤 在命令行窗口中使用 cd 指令，将工作文件夹切换到 GitStats 程序的文件夹。

3 步骤 确定 GitStats 程序输出网页文件的存储路径，并创建好该文件夹。

4 步骤 使用以下格式执行 GitStats 程序，其中“Git 文档库路径”就是我们程序项目的路径：

```
python gitstats.py "Git 文档库路径" "存储网页文件的路径"
```

读者可以参考以下范例：

```
python gitstats.py "D:\android projects\AndroidApp" "D:\gitstats
results"
```

执行成功之后，在指定的输出文件夹中会生成许多文件。用鼠标双击其中的 index.html 文件，就会在默认的网页浏览器中看到如图 6-2 所示的屏幕画面，里面会列出一些项目的基本数据。网页上方有 6 个灰色按钮，按序依次是“General”、“Activity”、“Authors”、“Files”、“Lines”和“Tags”。“General”就是首页的屏幕画面，单击其他按钮就会显示 Git 文档库的各种统计图表，像是在图 6-3 中单击“Activity”之后，显示项目在一天的不同时段 commit 次数的统计，图 6-4 是单击“Authors”之后，显示所有开发人员的 commit 次数统计表，以及每一天加入程序代码的行数累计图。GitStats 程序总共提供了超过 20 种图表，以便让我们了解 Git 文档库的状态。

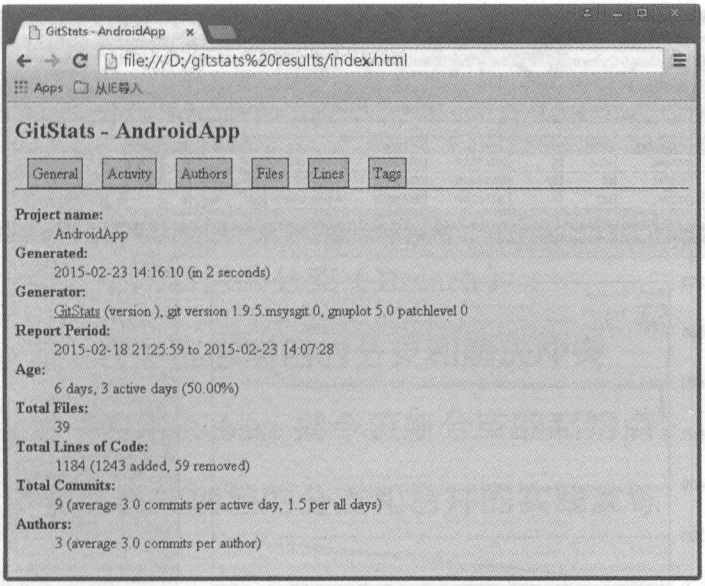


图 6-2　GitStats 程序产生的统计数据首页

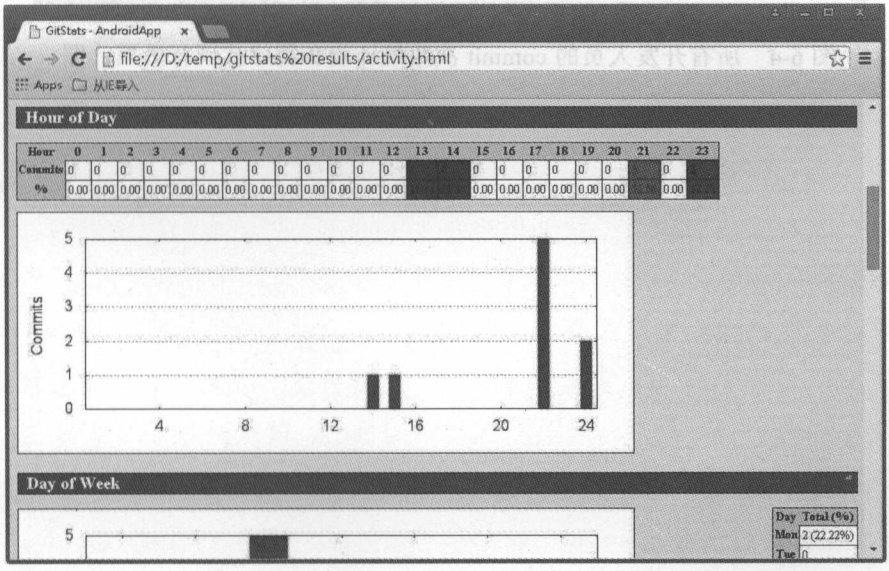


图 6-3　不同时段 commit 次数统计图表

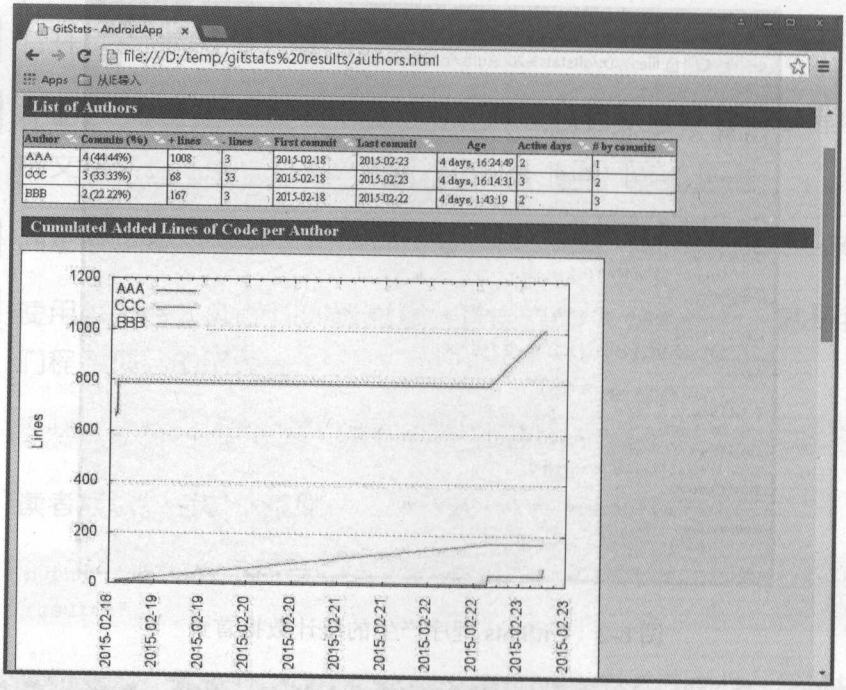


图 6-4　所有开发人员的 commit 次数统计表和程序代码行数累计图

第二部分　建立分支、合并和解决冲突

第 7 课　程序项目的分支（Branch）

第 8 课　合并程序项目的分支和解决冲突

第 9 课　使用 Rebase 指令更新分支的起始点

第 10 课　程序项目的分支和合并的实践演练



第 7 课

程序项目的分支（Branch）

开发程序项目的过程中，我们可能需要从原来的程序项目分出另一个程序项目来修改，这个操作称为 branch。分出来的程序项目可以和原来的程序项目再进行合并，或者形成两个独立的程序项目，要如何处理就看我们的需求而定。其实当执行“git init”指令的时候，Git 就会自动产生一个名字叫作 master 的主要分支。在 Git Bash 程序屏幕画面的提示字符上方，最后面的括号显示的，就是当前正在操作的分支名称，如以下格式：

```
用户名称@计算机名称 文件夹路径 (master)    ← master 表示当前正在操作 master 这个分支
$
```

如果启动 gitk 图形查看模式，左上方窗口的 commit 节点会有一个矩形框的标签（如图 7-1），里面显示的也是分支的名称。

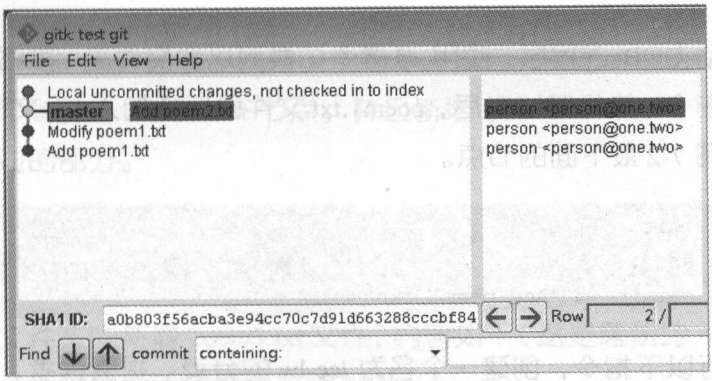


图 7-1　gitk 图形查看模式的 commit 节点标示出分支名称

在 Git 文档库中创建分支需要使用“git branch”指令，它的格式如下：

```
git branch 自己取的分支名称 commit 节点标识符或是标签
```

这个指令按照后面参数的多少，会有不同的功能：

1. 如果最后指定了 commit 节点标识符或是标签，就会从该节点“长”出分支；如果没有指定 commit 节点，就会从最新的 commit“长”出分支。

2. “git branch”指令后面没有接任何参数时，会列出当前文档库中正在开发的所有分支。

建立分支之后，就可以使用“git checkout 分支名称”这个指令来切换当前操作的分支。下面用一个简单的例子来示范，请按照以下步骤执行：

1 步骤 在某一个文件夹中创建一个名字叫作 poem1.txt 的文本文件，它的内容如下（这是唐朝诗人“李白”的《月下独酌》）：

> 花间一壶酒
> 独酌无相亲

2 步骤 启动 Git Bash 程序，使用 cd 指令切换到这个文件夹，然后执行以下指令。执行完毕之后，poem1.txt 文件已经被加入 Git 文档库中，如图 7-2 最下面的节点。

```
git init
git add .
git commit -m 'Add poem1.txt'
```

3 步骤 执行以下指令，创建一个名为 lee-by 的分支。提醒读者，虽然现在已经创建了 lee-by 这个分支，可是当前操作的分支还是 master。

```
git branch lee-by
```

补充说明

1. 如果要在创建分支之后，自动切换到新创建的分支，可以执行指令“git checkout -b 新分支的名称 commit 节点标识符或是标签”。这个指令等同于先执行“git branch 新分支的名称 commit 节点标识符或是标签”，接着再执行“git checkout 新分支的名称”。
2. 也可以指定要从特定的分支再长出另一个新的分支。例如创建 lee-by 分支之后，可以再执行“git branch lee-by2 lee-by”，就会新增 lee-by2 分支。
3. 分支的命名是按照创建该分支的目的，例如为了测试和修正 bug 所创建的分支，通常是用 bug 开头，后面加上斜线字符以及 bug 编号或是说明，如“bug/s001/crash-open-file”。为了测试新功能所创建的分支可以取名为“feat/multi-selection”。
4. 如果要显示特定名称的分支，可以使用“git branch”指令搭配“--list”选项，例如执行“git branch --list bug/*”会显示所有以“bug/”开头的分支。

4 步骤 现在在文件夹中新增一个名为 poem2.txt 的文件，内容如下（这是唐朝诗人“张继”的《枫桥夜泊》）：

> 月落乌啼霜满天
> 江枫渔火对愁眠

5 步骤 回到 Git Bash 程序，执行以下指令。执行完毕后，会在 master 分支中增加一个 commit 节点，并且新增 poem2.txt 文件，如图 7-2 左边的部分。

```
git add .
git commit -m 'Add poem2.txt'
```

6 步骤 在文件夹中新增 poem3.txt 文件，内容如下（这是唐朝诗人“李白”的《黄鹤楼送孟浩然之广陵》）：

```
故人西辞黄鹤楼
烟花三月下扬州
```

7 步骤 回到 Git Bash 程序，执行以下指令。执行完毕后，会在 lee-by 分支中增加一个 commit 节点，并且新增 poem3.txt 文件。

```
git checkout lee-by
git add .
git commit -m 'Add poem3.txt'
```

完成以上所有的步骤之后，就会得到图 7-2 的结果。

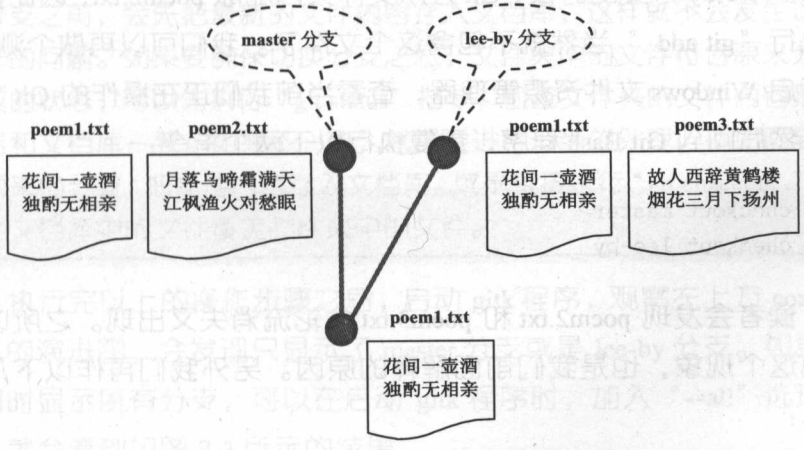


图 7-2　Git 文档库中的分支图

图 7-2 乍看好像是理所当然，没有什么问题。但是如果我们仔细

想一下：为什么 poem2.txt 没有包含在 lee-by 这个分支中？在前面的步骤 5 之后，我们并没有把这个文件从文件夹中删除，但是最后这个文件却从 lee-by 分支中消失了。这个问题牵涉到一个很重要的概念：当我们切换到另一个分支的时候，Git 会对比当前分支和将要切换过去的分支是否被 tracked 的文件列表有不一致的情况（请注意，是比较文件列表，不是文件内容）。如果有些文件被当前的分支 track，但是要切换过去的分支却没有 track，这些文件就会被删除，因为这些文件是后来才被加入的，并不属于将要切换过去的分支。相反的情况，如果有些文件当前的分支没有 track，但是要切换过去的分支却有 track，那么这些文件就会从文档库中取回，因为 Git 要让分支恢复它原来的状态。

了解 Git 的工作原理之后，读者是不是能够回答：为什么 poem2.txt 没有包含在 lee-by 这个分支中呢？关键就在前面步骤 7 的第一个指令“git checkout lee-by”，它是从 master 分支切换到 lee-by 分支。虽然我们把 poem2.txt 文件加入 master 分支，但是切换到 lee-by 分支的时候，原来在 lee-by 分支中并没有这个文件（参考图 7-2，lee-by 分支是从第一个 commit 节点“长”出来的，那时候还没有加入 poem2.txt）。所以切换到 lee-by 分支的时候，Git 会从文件夹内删除 poem2.txt，因此接下来执行“git add .”当然就不包含这个文件了。我们可以再做个测试，先开启 Windows 文件资源管理器，查看当前我们正在操作的 Git 文件夹，然后回到 Git Bash 程序，重复执行以下两个指令：

```
git checkout master
git checkout lee-by
```

读者会发现 poem2.txt 和 poem3.txt 会轮流消失又出现。之所以会出现这个现象，也是我们前面解释的原因。另外我们再作以下几点补充：

1. 在切换分支的时候，Git 会先对比文档库中当前分支的文件内容是否和将要切换过去的分支文件内容相同。针对内容不一样的文

件，Git 需要从文档库中取出该文件，这是为了让文件夹中的文件符合分支原来的状态。但是为了避免数据遗失，当 Git 要覆盖文件夹中的文件时，会检查该文件的内容是否已经加入文档库。如果还没有加入，Git 会显示警告信息，并且停止执行，以免资料遗失。如果确定不想保留这些已经修改，却还没有加入文档库的文件，可以换成执行“git checkout -f 分支名称”，Git 就会强制覆盖修改后的文件。

2. 除了被 tracked 的文件以外，对于 untracked 和 ignored 的文件，Git 都不会进行任何处理。

补充说明

虽然 Git 切换分支之后，会“尽可能”让文件夹中的文件恢复成原来该分支的状态，但是并不保证一定一样。例如当前分支和即将切换过去的分支都有文件 A，而且内容完全一样。但是我们在当前的分支中修改了文件 A，却没有 commit。这时候切换分支的时候，Git 对比文档库中两个分支的文件 A 是相同的，所以不会从文档库中取出文件 A，结果切换分支之后，文件 A 就不会回到原来的状态。这种错误的起因有一部分是操作上的瑕疵，因为在一般情况下，在切换分支之前，会先把最新的文件内容存入文档库，这样就不会发生这样的问题。如果要确保切换分支之后，文件夹中的文件符合原来分支的状态，可以先执行“git status”指令，检查文件夹的文件内容是否和文档库一致。如果不一致，就需要进行适当的处理，例如回到原来的分支，把文件内容存入文档库。或是直接执行“git checkout .”，用文档库中的文件覆盖文件夹中的文件。

执行完以上的操作步骤之后，启动 gitk 程序，观察左上方 commit 节点的演进图，会发现只显示了 master 分支或是 lee-by 分支。如果想要同时显示所有分支，可以在启动 gitk 程序时，加入“--all”选项如下，就会看到如图 7-3 所示的结果。

```
gitk --all
```

或是使用“git log”指令搭配以下选项：

```
git log --graph --oneline --all --decorate
```

“--graph”选项是画出 commit 节点的演进图、“--oneline”是用最精简的方式显示、“--all”表示要显示所有分支、“--decorate”是要标示分支的名称。执行以上指令后会显示类似下面的结果。“*”表示 commit 节点，右边会显示标识符的前 7 个数字。

```
* 7b16e52 (HEAD, lee-by) Add poem3.txt
| * 5aa497b (master) Add poem2.txt
|/
* 85d074e Add poem1.txt
```

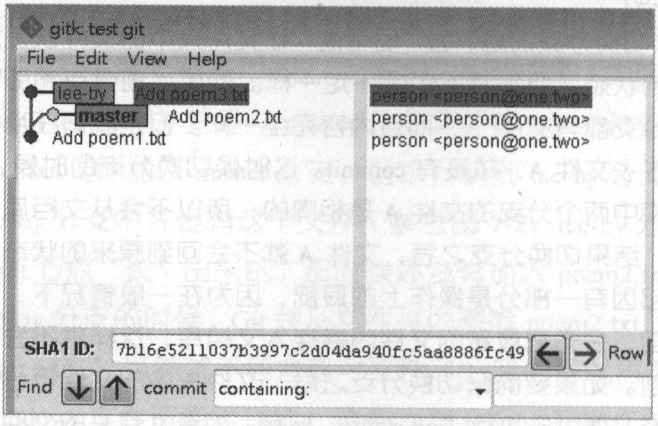


图 7-3　使用 gitk 图形查看模式查看 commit 节点的演进图

不管是在哪一个分支，我们都可以使用前面学过的指令和操作技巧来更新文档库中的文件，或是加入新文件，也可以删除 commit 节点。如果要删除分支，必须先切换到另一个分支，再执行以下指令：

```
git branch -d 要删除的分支名称
```

在一般情况下，分支应该合并到另一个分支。如果要删除还没有合并的分支，Git 会显示错误信息，并且停止删除分支的操作。如果确定要删除还没有合并的分支，可以把以上指令的“-d”选项换成“-D”，

要求 Git 强制执行删除分支的操作。如果想要更改分支的名称，必须先切换到该分支，然后执行以下指令：

```
git branch -m 新的分支名称
```

现在我们已经了解分支的概念，接下来要介绍一个操作 Git 时可能会发生的问题。第一次遇到这个问题时，通常不知道该如何处理，也不知道为什么会发生，现在让我们来解释一下。

7-1 解决 Detached HEAD 的问题



图 7-4 Git 文档库示意图

假设已经创建一个 Git 文档库，并且加入了一些 commit 节点，如图 7-4。现在执行以下指令：

```
git checkout HEAD^
```

Git 会显示以下信息，表示当前已经变成 detached HEAD 状态：

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by performing another checkout.
```

这时候 HEAD 是指向倒数第 2 个 commit 节点，如图 7-5 所示。另外请读者留意，Git Bash 程序的命令提示符的上一行，最后括号中已经不是分支名称，而是 commit 节点的标识符，表示现在已经不属于任何一个分支（提示：如果要回到某一个分支，只要执行“git checkout 分支名称”即可）。

补充说明

“git checkout commit 节点”和“git checkout commit 节点 .”这两个指令的执行结果有些不同。如果从文件夹的内容来看，这两个指令都会让文件夹中的文件变成指定的 commit 节点的版本。但是除此之外，第一个指令还会让当前分支的 HEAD 指向该 commit 节点，形成所谓的 detached HEAD 状态，但是第二个指令不会更改 HEAD 的状态。

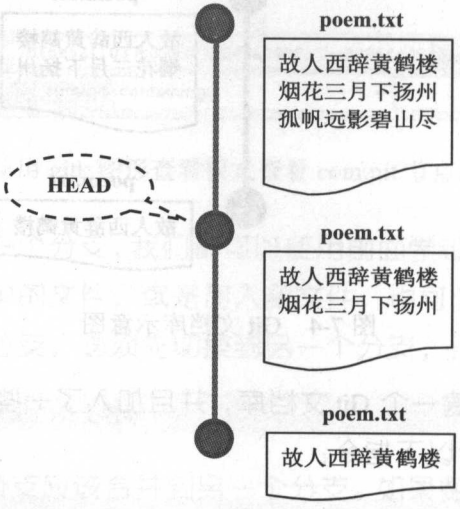


图 7-5　执行“git checkout HEAD^”之后的状态

现在假设修改文件夹中的文件，并且使用“git add”和“git commit”指令将修改后的文件加入文档库。这时候文档库就会变成如图 7-6 所示的状态，也就是从当前所在的节点，长出一个没有名称的分支。如果要查看这个无名的分支，必须执行“git log --graph --oneline --all --decorate”指令，“gitk --all”图形查看模式不会显示这个无名分支。如果执行“git status”指令，则会显示以下信息，这时候该如何处理呢？

```
HEAD detached from <commit 节点标识符>
```

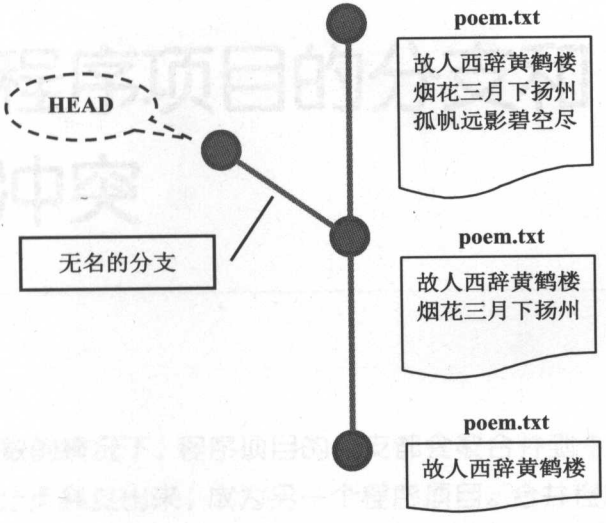


图 7-6　在 Detached HEAD 状态执行“git commit”指令的结果

我们有两种选择：第一是放弃这个无名分支的文件内容，也就是直接删除这个无名分支；第二是把这个无名分支的文件内容合并到原来的分支。第二个方法留到下一个单元再来介绍，我们先说明第一个解决办法。首先是赋予这个无名分支一个名称，然后切换到另一个分支，再将这个无名分支删除，也就是按序执行以下指令：

```
git branch 无名分支的分支名称    ← 帮这个无名分支取一个名称
git checkout master
git branch -D 无名分支的分支名称
```

执行完毕之后，这个无名分支就会从 Git 文档库中消失，文档库也会还原成原来的状态。





第 8 课

合并程序项目的分支和解决冲突

在大多数的情况下，程序项目的分支都会被合并到主要分支。除非决定将该分支独立出来，成为另一个程序项目。合并程序项目的分支需要使用“git merge”指令，它的格式如下：

```
git merge 分支名称
```

这个指令会把指定的分支，合并到当前 Git 工作中的分支。现在假设想要合并分支 A 和分支 B，如果执行以下指令，得到图 8-1 左边的结果。请读者注意，最上方合并后的 commit 节点属于分支 A。

```
git checkout A
git merge B
```

如果换成执行以下指令：

```
git checkout B
git merge A
```

得到图 8-1 右边的结果，最上方合并后的 commit 节点属于分支 B。分支被合并之后，名称还是会继续存在，我们必须将被合并的分支删除。假设已经把分支 B 合并到分支 A，接着可以执行“git branch –d B”把分支 B 删除。

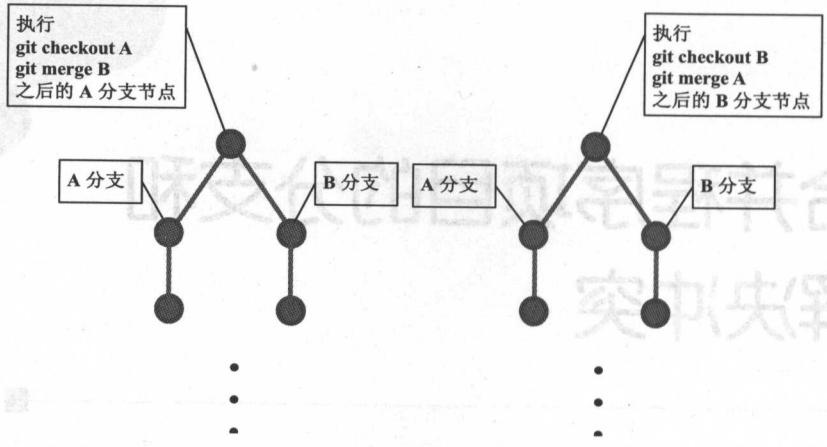


图 8-1　被合并的分支和工作中的分支关系图

补充说明

1. 如果执行合并之后想要反悔，可以使用前面单元学过的“git reset”指令。假设我们刚刚执行过合并分支的操作，现在想要恢复到未合并前的状态，可以执行以下指令：

```
git reset --hard HEAD^
```

“--hard”选项表示文件夹中的文件也要一起恢复。

2. 合并之后的 HEAD 节点会有两个父节点，可以分别用“HEAD^1”和“HEAD^2”来指定，或是直接使用 commit 节点的标识符来指定。

虽然合并分支是一个很简单的概念，可是在实际操作时，有一些不同的情况需要考虑。先来看看最简单的情况，在创建一个新的分支之后，我们只在这个新分支上进行修改和 commit，原来的分支没有做

任何改动，如图 8-2 左边的情况。bug/123 是从 master 长出来的分支，我们在 bug/123 分支上做了一些修改，并且执行两次 commit。在这一段时间之内，master 分支上没有进行任何改动。这时候如果要合并 master 和 bug/123 这两个分支，必须在 master 分支下执行“git merge bug/123”。如果换成在 bug/123 分支下执行“git merge master”，并不会得到合并的结果。我们可以把合并想象成“从被合并的分支中，复制新的文件内容到当前的分支”。因为新的文件内容是在 bug/123 分支，所以必须执行“git merge bug/123”，才会把新数据复制过来。合并 master 和 bug/123 分支之后，得到的是图 8-2 右边的结果。这种合并其实是把 bug/123 分支的修改完全运用到 master 分支。Git 把这种类型的合并称为 fast-forward merge，因为它就像是让 master 分支沿着 bug/123 分支“快进”。

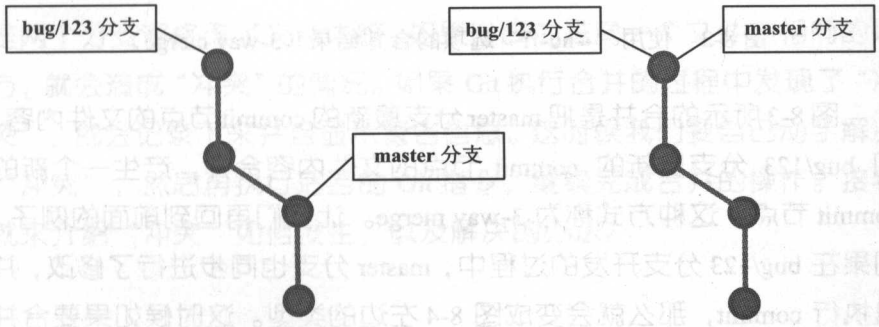


图 8-2　Fast-forward merge 合并前和合并后的状态图

Fast-forward merge 有一个特点，就是它不会产生新的 commit 节点。读者可以观察图 8-2，master 分支只是“快进”到 bug/123 分支的 HEAD，因此不会留下合并的记录。如果希望留下合并的记录，可以在执行合并时使用以下指令格式：

```
git merge --no-ff 分支名称
```

“--no-ff”选项表示不要使用 fast-forward merge。如果合并 master 和 bug/123 分支时使用这个选项，将会得到图 8-3 所示的结果。最上面的

commit 节点会自动产生，而且附带说明文字“Merge branch ‘bug/123’”。

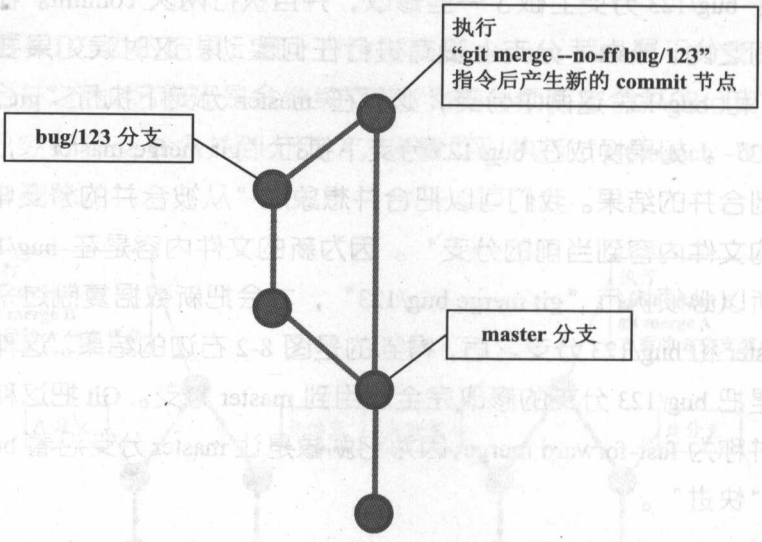


图 8-3　使用“--no-ff”选项的合并结果（3-way merge）

图 8-3 所示的合并是把 master 分支最新的 commit 节点的文件内容，和 bug/123 分支最新的 commit 节点的文件内容合并，产生一个新的 commit 节点，这种方式称为 3-way merge。让我们再回到前面的例子，如果在 bug/123 分支开发的过程中，master 分支也同步进行了修改，并且执行 commit，那么就会变成图 8-4 左边的类型。这时候如果要合并 master 和 bug/123 这 2 个分支，就不能使用 fast-forward merge，而必须采用 3-way merge。执行“git merge”指令的时候，Git 会自动判断应该使用 fast-forward merge 或是 3-way merge，除非加上“--no-ff”选项。执行合并之后，得到的结果就是图 8-4 右边的类型。

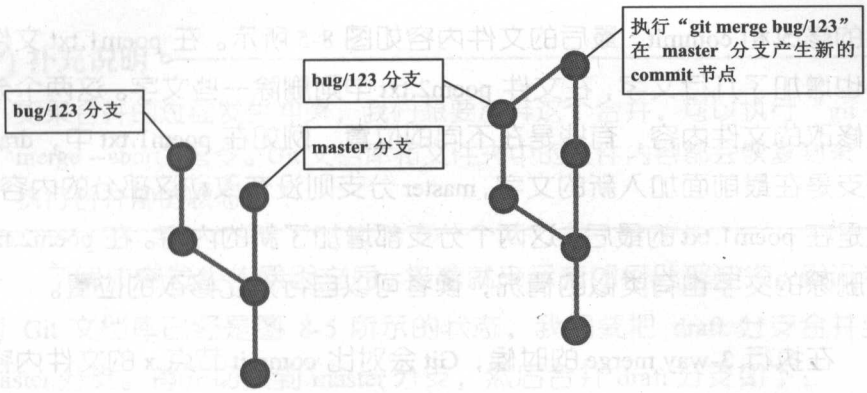


图 8-4　2 个分支同时进行修改的情况

3-way merge 的处理方式比 fast-forward merge 要复杂。因为 fast-forward merge 的文件内容只会在一个分支中修改，另一个分支没有任何变动，所以保证一定可以合并成功。但是 3-way merge 的情况是两个分支都修改了文件内容，如果修改的是同一个文件中相同的地方，就会造成“冲突”的情况。如果 Git 执行合并的过程中发现了“冲突”，就会记录下来并且显示警告信息。这时候我们要自己动手解决“冲突”，然后再执行适当的 Git 指令，继续完成合并的操作。接着就来介绍“冲突”如何发生，以及解决的办法。

8–1　解决合并分支时发生的冲突

当 Git 执行 3-way merge 的时候，可能会发生冲突，但是在什么情况下才会发生呢？下面用一个实例来说明，请读者参考图 8-5，draft 是从 commit 节点 x 长出来的分支。commit 节点 x 包含 poem1.txt 和 poem2.txt 这两个文件。draft 分支在经过一连串的修改和 commit 之后，这两个文件的内容如图 8-5 所示。首先在文件 poem1.txt 中添加了几行文字。为了容易区分，我们把新加入的文字用粗体和下划线标示出来。另外，在文件 poem2.txt 中则删除一些文字。master 分支也经过了一连

串的修改和 commit，最后的文件内容如图 8-5 所示。在 poem1.txt 文件中也增加了几行文字，在文件 poem2.txt 中则删除一些文字。这两个分支修改的文件内容，有些是在不同的位置，例如在 poem1.txt 中，draft 分支是在最前面加入新的文字，master 分支则没有改动这部分的内容。但是在 poem1.txt 的最后，这两个分支都增加了新的内容。在 poem2.txt 中删除的文字也有类似的情况，读者可以自行对比修改的位置。

在执行 3-way merge 的时候，Git 会对比 commit 节点 x 的文件内容和 draft 分支以及 master 分支的差异。commit 节点 x 的文件内容称为 base，如果 draft 分支和 master 分支，对于 base 版本的文件修改了不同的位置（像是 poem1.txt 和 poem2.txt 最前面的修改），这种情况不会造成冲突，所以 Git 在进行 merge 时会直接运用。会造成冲突的情况是两个分支都修改了文件中相同的位置，就像是 poem1.txt 和 poem2.txt 最后面的部分。如果修改了文件相同的位置，但是修改的内容却不一样，就会造成冲突。

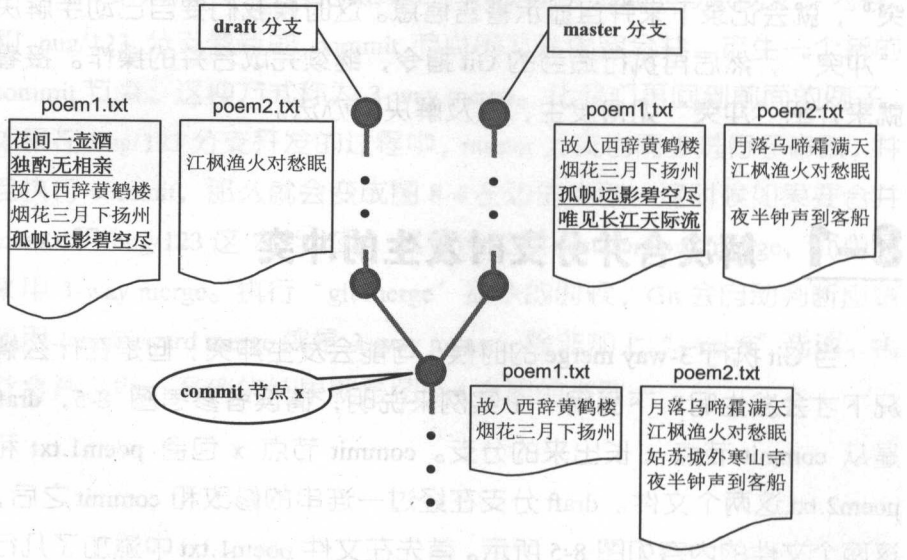


图 8-5　3-way merge 发生冲突的范例

补充说明

如果合并的过程发生冲突，我们想要放弃这个合并，可以执行“git merge --abort”指令。Git 文档库和文件夹中的文件内容都会恢复到未执行合并前的状态。

了解冲突发生的原因之后，接着就来示范如何处理冲突。假设当前 Git 文档库已经是图 8-5 所示的状态，我们要把 draft 分支合并到 master 分支。首先切换到 master 分支，然后合并 draft 分支如下：

```
git checkout master
git merge draft
```

Git Bash 程序屏幕画面会显示以下信息，表示有冲突发生。命令提示符上方的文件夹状态栏最后面的括号显示 MERGING，表示现在还处于合并的过程：

```
Auto-merging poem2.txt
CONFLICT (content): Merge conflict in poem2.txt   ← poem2.txt 在合并的过程中发生冲突

Auto-merging poem1.txt
CONFLICT (content): Merge conflict in poem1.txt   ← poem1.txt 在合并的过程中发生冲突
Automatic merge failed; fix conflicts and then commit the result.

用户名称@计算机名称 文件夹路径 (master|MERGING)   ← 表示当前正处于合并的过程
$
```

如果现在执行“git status”指令，会显示如下的操作说明：

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")   ← 要先解决冲突之后才能执行“git commit”指令

Unmerged paths:
  (use "git add <file>..." to mark resolution)   ← 解决文件的冲突之后执行“git add”指令
```

```
        both modified:   poem1.txt
        both modified:   poem2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

现在我们必须打开有冲突的文件，找到发生冲突的位置，然后加以修改。Git 会用连续 7 个“<”字符标示出发生冲突的位置。首先打开 poem1.txt 文件，会看到以下内容：

```
花间一壶酒
独酌无相亲
故人西辞黄鹤楼
烟花三月下扬州
孤帆远影碧空尽
<<<<<<< HEAD
唯见长江天际流
=======
>>>>>>> draft
```

如同前面的解释，冲突就发生在文件的最后，因为 master 和 draft 分支同时修改了这个位置的内容。我们只要修正冲突的内容，然后删除冲突的标示即可，也就是<、=和>开头的那几行。接着打开 poem2.txt 文件，它的内容如下：

```
江枫渔火对愁眠
<<<<<<< HEAD
夜半钟声到客船
=======
>>>>>>> draft
```

同样先找到发生冲突的位置，然后修正内容，再删除冲突的标示，最后将文件存储。修正完所有冲突的文件之后，可以执行“git diff”指令确认。如果没有显示任何差异，表示已经解决所有的冲突。最后执行以下指令完成分支的合并，并且删除被合并的分支。最后在 master 分支会创建一个新的 commit 节点。

```
git add .
git commit -m '操作说明'
git branch -d draft
```

这一小节我们解释了合并分支时发生冲突的原因和解决方法。虽然步骤并不困难，可是在修改冲突内容的时候，需要自行启动编辑程序，然后加载文件。有时候可能需要参考不同分支的文件版本，实际操作起来有点麻烦。这个问题可以使用 Merge Tool 来解决，下一小节我们就来学习相关的设置和用法。

8-2 设置Merge Tool解决合并分支时发生的冲突

在实践中要解决合并分支时发生的冲突，其实有更方便的做法。还记得我们在第 2 单元介绍的文件对比程序 KDiff3 吗？这一类的文件对比程序，通常还包含文件合并的功能，由于它们是使用图形模式操作，因此使用起来更方便。最棒的是 Git 能够和这些文件对比程序合作，让我们更方便地解决合并时发生的冲突。要使用外部程序解决合并时发生的冲突，可以在 Git 的配置文件加入 mergetool 相关的设置项，设置方法类似于第 2 单元介绍的 difftool。例如以下范例是指定用 KDiff3 程序来处理合并时发生的冲突：

```
git config --global merge.tool kdiff3
git config --global mergetool.kdiff3.cmd \
"'D:/Program    Files/KDiff3/kdiff3.exe'    \"\$BASE\"    \"\$LOCAL\"
\"\$REMOTE\" -o \"\$MERGED\""
git config --global mergetool.prompt false
git config --global mergetool.kdiff3.trustExitCode true
git config --global mergetool.keepBackup false
```

其中的 prompt 设置项是设置 Git 在启动外部程序之前，是否要询问我们；trustExitCode 是设置外部程序结束之后，Git 是否直接用它的返回值来判断合并是否成功；keepBackup 设置项决定是否要备份原来含有

冲突标示的文件。

补充说明

KDiff3 程序会用自己的设置项来决定是否要备份原来含有冲突标示的文件，该设置项是在主菜单的 Settings > Configure KDiff3 > Directory，其中有一个名为 Backup files (.orig) 的设置目，它就是用来设置是否要备份文件。

当我们在 Git Bash 程序的命令行执行上述指令之后，就完成了 Merge Tool 的设置。现在来测试一下 Merge Tool 的效果。首先执行以下指令，重置 Git 文档库，回到还没合并 draft 分支前的状态：

```
git reset --hard HEAD^
```

接下来按照前面的做法，开始合并 draft 分支：

```
git checkout master
git merge draft
```

Git 一样会报告发生了冲突。接下来就是 Merge Tool 上场的时候。我们执行以下指令，启动 Merge Tool：

```
git mergetool
```

图 8-6 是启动 KDiff3 程序作为 Merge Tool 的屏幕画面，左上方会显示 base 版本的文件，也就是 commit 节点 x 的文件。右边显示的是两个分支的文件，下方是 KDiff3 程序帮助我们完成合并的结果。我们可以自行修改这个合并的结果，完成之后保存文件，然后关闭 KDiff3 程序。由于有两个冲突的文件，Git 会再度启动 KDiff3 程序，打开第二个冲突的文件让我们处理。处理好所有冲突的文件之后，就会回到 Git Bash 程序，屏幕画面会显示以下信息：

```
Merging:
poem1.txt
poem2.txt
```

```
Normal merge conflict for 'poem1.txt':
  {local}: modified file
  {remote}: modified file

Normal merge conflict for 'poem2.txt':
  {local}: modified file
  {remote}: modified file
```

现在我们已经将合并的冲突处理完毕，并且 Git 已经自动把修改后的文件内容加入索引，也就是 Git 已经帮我们执行了“git add”指令。最后只要再执行下列指令，就可完成分支的合并。

```
git commit -m '操作说明'
git branch -d draft
```

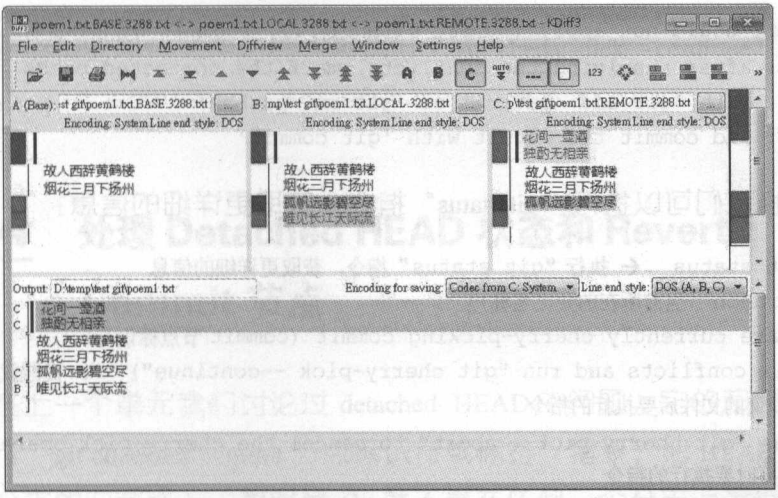


图 8-6　启动 KDiff3 程序作为 Merge Tool 的屏幕画面

8-3 使用 Cherry-Pick 指令运用指定的 Commit 节点版本

Merge 指令是把指定的分支，合并到当前所在的分支。Cherry-Pick

指令其实也是一种合并。只是它处理的对象不是分支，而是 commit 节点。也就是说，我们是把某一个 commit 节点的文件版本，合并到文件夹中的文件。它的指令格式如下：

```
git cherry-pick commit 节点标识符或标签
```

在默认情况下，执行这个指令会创建一个新的 commit 节点。如果不想要创建新节点，可以加上“-n”选项。另外要注意的是，执行这个指令之前，文件夹中被修改的文件必须先存入 Git 文档库，否则会出现警告信息，提醒我们必须把修改的文件存入文档库。由于 Cherry-Pick 指令需要执行合并，因此也会发生文件内容冲突的情况。如果出现这种情况，Git Bash 程序屏幕画面会显示以下信息：

```
error: could not apply (commit 节点标识符)... (commit 节点操作说明)
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

这时候我们可以执行“git status”指令，获取更详细的信息：

```
$ git status  ← 执行“git status”指令，获取更详细的信息
On branch (当前所在的分支名称)
You are currently cherry-picking commit (commit 节点标识符).
  (fix conflicts and run "git cherry-pick --continue")  ← 提醒我们编辑好有冲突的文件后要执行的指令
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)
← 放弃时要执行的指令

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   app/src/main/java/com/game/MainActivity.
java  ← 发生冲突的文件

no changes added to commit (use "git add" and/or "git commit -a")
```

发生冲突时我们有两个选择：一是放弃执行，回到原来的状态；二是编辑发生冲突的文件，再继续未完成的操作。如果决定要放弃，可以

直接执行“git cherry-pick --abort”，文件夹中的文件和 Git 文档库都会恢复到原来的状态。如果想要继续执行，必须打开发生冲突的文件，找到发生冲突的位置，然后加以修改。做法和前面小节介绍的方式完全相同。编辑好有冲突的文件之后，执行下列指令，就可以完成后续操作。

```
git add .
git cherry-pick --continue
```

另外也可以使用外部的 Merge Tool 来编辑有冲突的文件。只要在 Git Bash 显示冲突信息之后，执行以下指令，启动 Merge Tool：

```
git mergetool
```

处理好所有的冲突文件之后，再执行“git cherry-pick –continue”指令即可。

8–4 处理 Detached HEAD 状态和 Revert Commit 节点

在上一个单元我们讨论过 detached HEAD 的问题，它的起因是我们执行“git checkout commit 节点标识符或标签”指令，把 HEAD 标签移到分支的旧节点上。这时候 Git 就不是在任何一个分支上运行了。如果在这种状态下修改文件，并且执行 commit，就会从当前所在的节点，长出一个没有名称的分支。上一个单元我们介绍的解决方法是删除这个分支。现在我们已经学会合并分支的技巧，就可以把这个无名的分支，合并到原来的分支。第一步是帮这个无名的分支取一个名称，然后切换到原来的分支，再将它合并进来，执行的指令如下：

```
git branch 无名分支的分支名称    ← 帮这个无名分支取一个名称
git checkout master
```

```
git merge 无名分支的分支名称
```

这种合并属于 3-way merge，因此在合并的过程中可能发生冲突。解决冲突的方法就如同前面小节的介绍，解决冲突后的处理方式也完全相同。

我们已经学过了“git reset”指令，它可以将 Git 文档库恢复到某一个旧节点的状态。如果加上“--hard”选项，文件夹中的文件也会一起恢复。另外还有一个功能类似的 revert 指令，也可以让文件夹中的文件回到某一个 commit 节点的状态。不同的是，revert 指令不会删除 Git 文档库中的 commit 节点。相反地，在修改文件夹中的文件之后，它会新增一个 commit 节点。例如以下指令会让文件夹中的文件恢复到“HEAD^”这个节点的状态，而且在文档库中新增一个 commit 节点：

```
git revert HEAD
```

注意“git revert commit 节点”指令是回到“指定的 commit 节点的前一个节点”的文件状态。“git reset --hard commit 节点”则是回到“指定的 commit 节点”的文件状态。

补充说明

执行“git revert”指令会新增 commit 节点，因此会自动启动文字编辑程序让我们输入操作说明。Git 默认的文本编辑器是 vi（离开 vi 编辑器的方法是先输入一个分号“:”，光标会跳到屏幕左下方，接着输入 q，再按下 Enter 键即可），如果想要改用其他编辑器，可以参考第 2 单元的说明。

在执行“git revert”指令的时候，可能会碰到发生冲突的情况，可以使用前面小节介绍的方法来处理。如果在过程中反悔了，想取消 revert 指令，可以执行“git revert --abort”。其实“git revert”指令的功能类似“git checkout commit 节点 文件 1 文件 2...”，它们都是让文件夹中的文件恢复到某一个旧 commit 节点的状态，但是“git checkout”指令不会导致冲突的情况。要使用哪一个指令就看操作者的需求而定。



第 9 课

使用 Rebase 指令更新分支的起始点

现在我们已经学会如何帮程序项目创建分支，等到该分支开发完成，并且通过测试之后，再将它合并到主分支。这项功能可以让我们随心所欲地在分支中修改和测试程序，不需要担心影响到原来主分支的程序项目。接下来我们要再考虑另一种情况，如果创建程序项目的分支之后，这个分支需要比较长的时间才能够完成开发。另一方面，程序项目的主分支又持续地更新。在这种情况下，为了让分支中的程序代码能够和主分支同步，也就是维持二者基础程序架构的一致性，我们必须定时将主分支的修改合并到当前正在开发中的分支。这种合并和前面单元介绍的合并操作方式正好相反，前面单元的做法是在分支开发完成之后，将分支合并到主分支。这里的做法却是反过来，将

主分支的修改，合并到开发中的分支。如果以图 9-1 为例，就是执行以下指令：

```
git checkout draft
git merge master
```

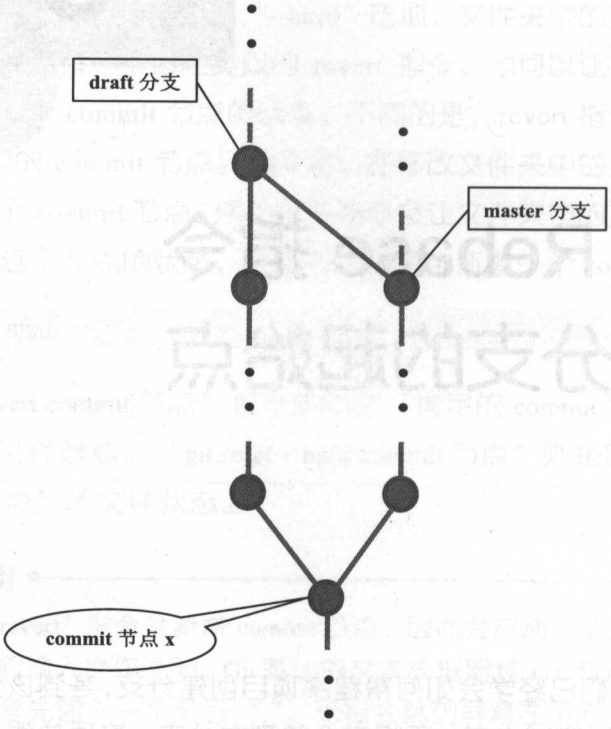


图 9-1　把 master 分支的修改合并到开发中的 draft 分支

执行合并之后，master 分支和 draft 分支又会持续更新，形成图 9-1 上方虚线的部分。一段时间之后，必须再度将主分支的修改合并到 draft 分支。如果这种情况持续下去，不难想象，最后 master 分支和 draft 分支会变成互相交织的网状形态。这样的结果不能算错，因为 Git 文档库依然可以正确地工作，程序项目的状态也完全正常，只是程序项目的 commit 节点演进图就会变得比较复杂。如果要让演进图更简单一些，我们可以用 rebase 指令取代 merge 指令。rebase 指令的功能也是

把另一个分支的修改运用到当前的分支，但是它的执行结果会变成如图 9-2 所示的状态。也就是说，原来从 commit 节点 x 长出来的 draft 分支，会变成从 master 分支最新的 commit 节点长出来。然后 master 和 draft 分支依然可以各自进行修改，一段时间之后，再度用 rebase 指令将 master 分支的修改运用到 draft 分支。直到 draft 分支全部完成之后，再使用 merge 指令将它合并到主分支。比较图 9-1 和图 9-2 的差别，很容易就可以发现，rebase 指令可以避免两个分支形成相互交织的情况，让程序项目的 commit 节点演进图变得比较简单，而且容易理解。

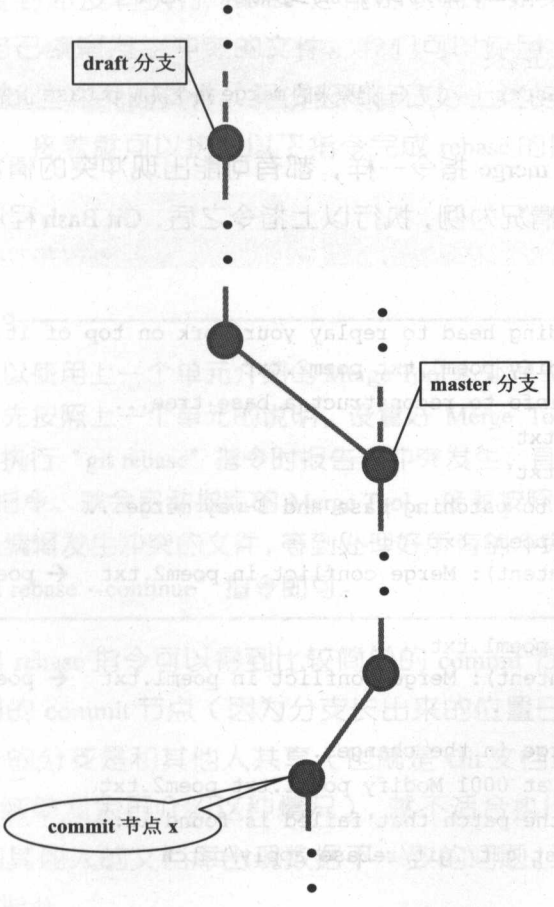


图 9-2　使用 rebase 指令将 master 分支的修改运用到开发中的 draft 分支

9-1 Rebase 指令的用法

了解 rebase 指令的功能之后，接着就来介绍它的用法。其实很简单，只要把原来使用 merge 指令的地方，换成使用 rebase 指令即可。例如以下是用 merge 指令把 master 分支的修改合并到 draft 分支：

```
git checkout draft
git merge master
```

如果换成使用 rebase 的方式，就变成以下指令：

```
git checkout draft
git rebase master          ← 把原来的 merge 指令换成 rebase 指令
```

rebase 指令和 merge 指令一样，都有可能出现冲突的情况。用前一个单元图 8-5 的情况为例，执行以上指令之后，Git Bash 程序会显示以下信息：

```
First, rewinding head to replay your work on top of it...
Applying: Modify poem1.txt poem2.txt
Using index info to reconstruct a base tree...
M       poem1.txt
M       poem2.txt
Falling back to patching base and 3-way merge...
Auto-merging poem2.txt
CONFLICT (content): Merge conflict in poem2.txt   ← poem2.txt 文件发生
冲突
Auto-merging poem1.txt
CONFLICT (content): Merge conflict in poem1.txt   ← poem1.txt 文件发生
冲突
Failed to merge in the changes.
Patch failed at 0001 Modify poem1.txt poem2.txt
The copy of the patch that failed is found in: .
  d:/temp/test git/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".   ←
解决冲突后的处理方式
```

```
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".

用户名称@计算机名称 文件夹路径(draft|REBASE 1/1)  ← 括号中显示 REBASE 表示当前还处于 rebase 的过程
$
```

注意，命令行提示字符上方状态栏的最后会显示 REBASE，表示当前还处于 rebase 的状态。如果我们决定要放弃这一次的 rebase 操作，可以在还没有完成 rebase 之前，执行“git rebase --abort”指令，Git 文档库就会恢复到还没有执行 rebase 之前的状态。如果要继续执行 rebase，就要自己编辑发生冲突的文件。我们可以使用上一个单元介绍的方法处理这些冲突的文件，假设已经修改好上述的 poem1.txt 和 poem2.txt 文件，接着就可以执行以下指令完成 rebase 的操作：

```
git add .
git rebase --continue
```

补充说明

我们同样可以使用上一个单元介绍的 Merge Tool 来处理 rebase 发生的冲突，只要先按照上一个单元的说明，设置好 Merge Tool 相关的设置项。如果执行“git rebase”指令时报告有冲突发生，直接执行“git mergetool”指令，就会启动指定的 Merge Tool。接着按照上一个单元介绍的方法编辑发生冲突的文件，等到处理好所有的冲突文件之后，再执行“git rebase --continue”指令即可。

虽然使用 rebase 指令可以得到比较简单的 commit 节点演进图，但是它会改动旧的 commit 节点（因为分支长出来的位置已经不一样）。如果当前工作的分支是和其他人共享（也就是 Git 文档库是在远程的服务器上，后续单元会再介绍这种情况），就不适合执行 rebase 指令，否则会造成和其他人的文档库出现数据不一致的问题。这种情况下应该使用 merge 指令。

9-2 执行 Rebase 之后想要反悔怎么办？

如果执行 rebase 指令之后发生冲突，Git 会停下来让我们处理冲突的文件，这时候如果要放弃 rebase，可以执行“git rebase --abort”。可是如果 rebase 指令已经从头到尾执行完毕，这时候想要恢复到 rebase 前的状态该怎么办？没问题，我们可以使用“git reset”指令，但是必须先找到执行 rebase 指令之前，HEAD 所在的 commit 节点。由于 HEAD 代表当前工作中的分支之最新的 commit 节点，它会随着我们的操作不断地变动，要如何找到旧的 HEAD 的数据呢？Git 提供一个“git reflog”指令，可以让我们查询 HEAD 或是任何分支变动的历史记录，这个指令的格式如下：

```
git reflog HEAD或是任何分支的名称
```

例如执行“git reflog master”会显示类似如下的结果：

```
6bb0d3f master@{0}: commit: Modify poem2.txt.
0c1cf0d master@{1}: commit: Modify poem1.txt poem2.txt
09df545 master@{2}: commit (initial): Add poem1.txt, poem2.txt
```

最上面那一行是当前 master 分支最新的 commit 节点，也就是说在当前的状态下，我们可以用“master@{0}”表示 6bb0d3f 这个 commit 节点。但是，如果我们又对 master 分支进行一项新的操作，它们的对应关系就会变动，这时候就要重新执行“git reflog master”才能得到新的对应关系。“git reflog”指令显示的结果是按照时间顺序排列，越早以前的排在越下面，我们可以使用“master@{数字}”来表示它所对应的 commit 节点。

如果执行“git reflog”指令的时候不加任何参数，就会列出 HEAD 变动的历史记录。假设已经按照前一个小节的操作步骤完成 rebase，接着执行“git reflog”指令，就会看到类似如下的结果：

```
a161958 HEAD@{0}: rebase finished: returning to refs/heads/draft
a161958 HEAD@{1}: rebase: Modify poem1.txt poem2.txt
0c1cf0d HEAD@{2}: rebase: checkout master
e54844b HEAD@{3}: checkout: moving from master to draft
…(更早以前的 HEAD 记录)
```

从最上面三行的文字信息可以看出它们都是执行 rebase 所导致的变动。也就是说，在执行 rebase 指令之前，HEAD 所在的 commit 节点是在“HEAD@{3}”这个位置。找到这个节点的位置之后，只要执行以下指令，就可以让 Git 文档库回到 rebase 之前的状态：

```
git reset --hard HEAD@{3}
```



第10课

程序项目的分支和合并的实践演练

学会使用 Git 指令和操作方法之后，在这个单元我们要介绍如何在开发程序项目的过程中，导入 Git 管理机制。首先要介绍的是使用 Git 的策略，接着再用一个实际的程序项目来示范。

10–1 程序项目开发导入 Git 的策略

首先，回想一下开发程序项目的过程。刚开始的时候，我们会用开发工具创建一个程序项目，然后按照用户的需求，开始编写程序代码以及准备需要的项目资源文件。基本上，我们会按照用户对于程序功能的需求，来区分不同的开发阶段。例如先完成功能 1，再开发功能 2，依此类推。等到所有的功能都完成之后，就可以交付给用户，

甚至开始着手开发下一个版本。这整个过程可以用图 10-1 表示。

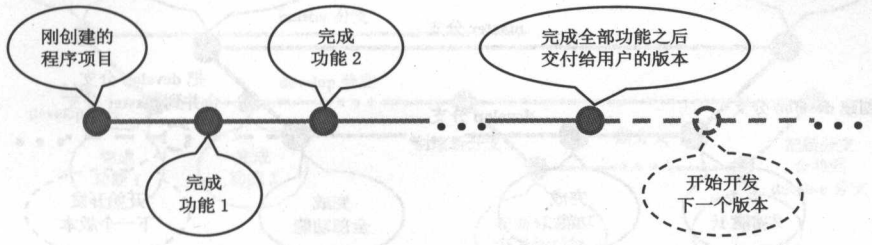


图 10-1　程序项目的开发过程

图 10-1 和我们已经学过的 Git 文档库的 commit 节点演进图非常类似。也就是说，可以使用 Git 指令，记录程序项目开发过程中的每一个阶段。而且日后我们可以随时获取每一个阶段的程序代码。虽然把图 10-1 直接当成 commit 节点的演进图似乎是理所当然，但是如果再从细节来思考，就会发现有些情况并不是那么理想。例如在交付给用户第一版的程序之后，又继续开发下一个版本。日后如果想要取出第一版的程序代码，必须在图 10-1 的 commit 节点中，找到正确的节点。再者，交付给用户的版本应该比较重要，如果把它掺杂在开发中的版本中，会有比较大的风险。因为 commit 节点有可能不小心被 reset，一切就前功尽弃！

以上讨论的问题可以用 Git 文档库的分支来解决。也就是说，我们把正式版的程序项目和开发中的版本，放在不同的分支。例如原来的 master 分支专门保存正式版的程序项目，另外创建一个 develop 分支，则用来保存开发中的版本。如此一来，就会变成图 10-2 的架构。如果采用这种开发模式，可以很容易地在 master 分支上，找到程序项目的所有正式版本。而且绝大部分的时间都在 develop 分支上工作，只有完成正式版的时候，才需要把结果合并到 master 分支。因而可以有效降低因为操作疏失，而损坏正式版程序项目的可能性。

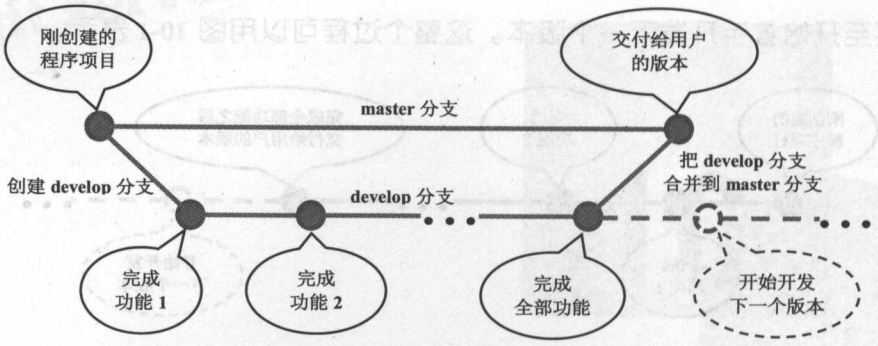


图 10-2　使用分支保存正式版和开发中的程序项目

在 develop 分支上开发程序还可以再细分成两种情况。第一种是程序修改的幅度不大，需要的时间不会太久，短短几天之内一定可以完成。这时候可以直接在 develop 分支上修改，完成之后再 commit。第二种情况是程序修改的幅度比较大，所需的时间长短比较难掌握，甚至不确定最后是否会成功。这时候可以从 develop 分支再长出另一个分支，例如可以将这个新分支取名为 feature/new-function-name，然后切换到这个新分支，开始着手修改程序。在这个新分支上，我们可以放心地进行修改，执行 commit，完全不用担心会影响到 develop 分支上的程序项目。如果开发的过程中，develop 分支上的程序代码又被更新，可以随时使用 rebase 指令，将这个新分支的程序代码进行同步更新。等到这个新分支的功能开发完成之后，再将它合并到 develop 分支。这种开发模式会形成图 10-3 的架构。如果最后决定不要采用新分支所开发的功能，可以在还没有合并之前，先切换到 develop 分支，再把这个新分支删除即可。了解如何将 Git 应用到程序项目的开发之后，接下来用一个实际范例，让读者亲身体会一下 Git 强大的程序代码管理能力。

补充说明

如果合并分支之后，想要回到未合并之前的状态，可以使用“git reset”指令，详细做法请参考第 8 课的介绍。

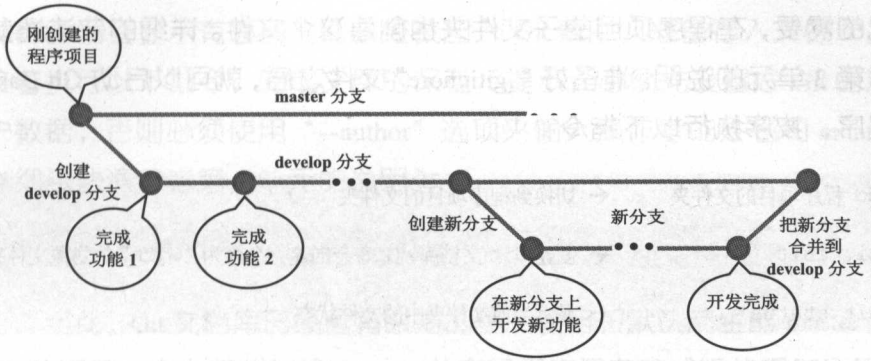


图 10-3　在 develop 分支上创建另一个分支，进行新功能的开发

10–2　结合 Git 与程序项目开发

决定 Git 的使用策略之后，我们就可以开始动手，实际体验一下将 Git 导入程序项目开发的完整过程。以下将程序项目分成“起始阶段”、“开发阶段”和“完成阶段”，说明如何运用 Git 指令来创建 Git 文档库、操作分支、执行 commit……等操作。

程序项目的“起始阶段”需要完成以下工作：

1. 创建一个新的程序项目。
2. 创建这个新项目的 Git 文档库。
3. 编辑“.gitignore”文件，排除不需要存入 Git 文档库的文件和文件夹，然后把原始的程序项目存入 Git 文档库。
4. 创建开发程序项目专用的分支。

有些程序开发工具在创建程序项目的时候，会自动生成“.gitignore”文件（例如 Android Studio）。如果程序开发工具没有生成“.gitignore”文件，我们就要自己使用文字编辑程序来创建这个文件。并且根据情

况的需要，在程序项目的子文件夹也创建这个文件。详细的做法请参考第 3 单元的说明。准备好“.gitignore”文件之后，就可以启动 Git Bash 程序，按序执行以下指令：

```
cd 程序项目的文件夹      ← 切换到程序项目的文件夹
git init                 ← 创建 Git 文档库，Git 会创建一个名为“.git”的隐藏文件夹
git status               ← 显示当前文件夹中的文件状态
```

执行“git status”指令之后，屏幕画面会显示有哪些文件和子文件夹是 untracked 的状态。例如以下是 Android Studio App 项目的执行结果，名称最后以斜线字符结尾的就是文件夹：

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        .idea/
        Game.iml
        app/
        build.gradle
        gradle.properties
        gradle/
        gradlew
        gradlew.bat
        settings.gradle

nothing added to commit but untracked files present (use "git add" to
track)
```

继续执行以下指令：

```
git add .
git status
```

屏幕画面会显示有哪些文件已经被加入 Git 索引，准备存入文档库。接着执行以下指令，完成文件的保存（提示：必须先设置好 Git 的用户数据，否则必须使用“--author”选项来输入操作者的姓名和 email，详细做法请参考第 2 单元的说明）：

```
git commit -m 'Initial app project.'
```

至此，Git 文档库已经含有原始的程序项目和默认产生的 master 分支。按照前一个小节的讨论，master 分支只用来保存正式版的程序项目。因此现在我们要创建一个开发程序项目使用的分支，请执行以下指令：

```
git checkout -b develop
```

这个指令会创建一个名为 develop 的分支，并且把当前工作中的分支，从 master 切换到新创建的分支。Git Bash 程序提示字符上方状态栏最后的括号内会显示 develop，表示当前是在 develop 分支中操作。现在如果执行“gitk”指令启动图形查看模式，会在屏幕画面左上方看到如图 10-4 所示的结果。表示现在有一个 develop 分支、一个 master 分支和一个 commit 节点。这样就完成了“起始阶段”的工作。

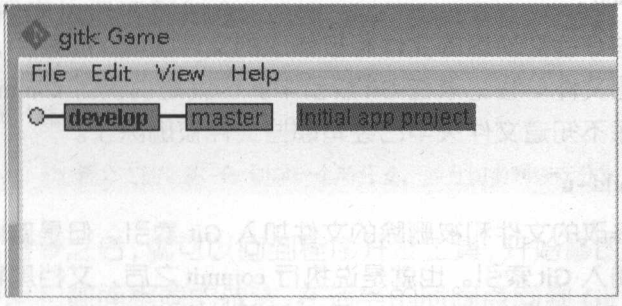


图 10-4　程序项目“起始阶段”的 Git 文档库状态

接下来就是在 develop 分支上进行程序项目的开发。我们可以切换到程序开发工具，然后按照惯用的方式编辑程序代码，加入、修改或是删除项目资源文件，以及进行任何开发程序项目的操作。然后在

适当的时间点，回到 Git Bash 程序，执行以下指令，把当前的成果送进 Git 文档库保存：

```
git add -A
git status
git commit -m '操作说明'
```

以上的“git status”指令不是必要的步骤。但是建议每一次执行“git add”指令之后，都执行“git status”指令，查看究竟有哪些文件被加入到 Git 索引中，以确保当前程序项目的状态被正确地送进文档库保存。不会因为操作的疏失，而有任何遗漏。另外，Git Bash 程序不需要一直保持在执行状态，我们可以在操作 Git 文档库告一段落的时候，例如“起始阶段”完成时，结束 Git Bash 程序。等到下次要执行 Git 指令的时候，再重新启动它，并且切换到程序项目的文件夹。Git Bash 程序会自动恢复到前一次结束时的状态。

补充说明

“git add”指令搭配不同的选项和参数，会把不同状态的文件（例如被修改的文件、新增的文件、被删除的文件）加入到 Git 索引中：

1. “git add .”

 这个指令会把新增的文件和被修改的文件加入 Git 索引，但是被删除的文件不会记录在 Git 索引中。也就是说执行 commit 之后，文档库不知道文件夹中已经有哪些文件被删除了。

2. “git add -u”

 把被修改的文件和被删除的文件加入 Git 索引，但是新增的文件不会加入 Git 索引。也就是说执行 commit 之后，文档库将不会包含新增的文件，但是会停止更新已经被删除的文件。

3. “git add -A”

 把新增的文件、被修改的文件以及被删除的文件全部加入 Git 索引。执行 commit 之后，文档库将会包含修改的文件和新增的文件，而且会停止更新已经被删除的文件。

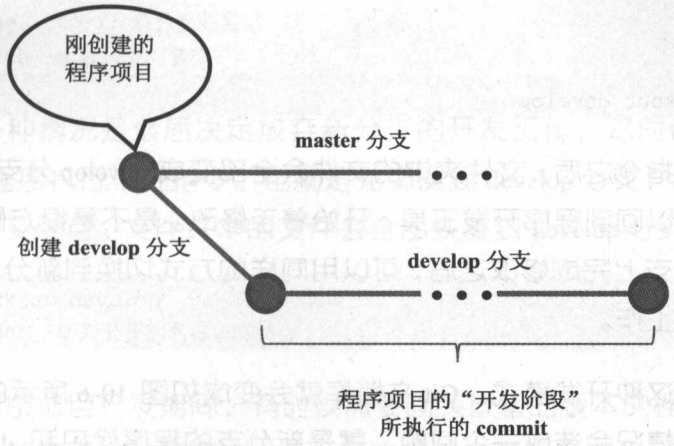


图 10-5　程序项目在“开发阶段”的 Git 文档库形态

按照以上讨论的开发模式，经过几次 commit 之后，Git 文档库就会变成如图 10-5 所示的状态。这种开发方式适合用于短时间（例如 2~3 天）就可以完成的情况。而且在这段时间内，不会因为其他原因，需要对程序项目的其他部分进行修改。如果程序项目的修改或是新功能的开发，需要比较长的时间才能完成。这种情况应该从 develop 分支延伸出另一个分支，然后在这个新分支上进行开发工作。我们可以在 Git Bash 程序执行下列指令：

```
git checkout develop          ← 切换到 develop 分支，如果现在已经在 develop 分支，可以略过这个指令

git checkout -b 新分支的名称 ← 创建一个新分支，并且切换到该新分支
```

执行完上述指令之后，就可以回到程序开发工具，开始修改程序项目。我们可以使用和前面相同的做法，在适当的时间点回到 Git Bash 程序，把当前的状态存入 Git 文档库。

这种在新分支上修改程序的做法，可以随时回到 develop 分支，修改原来的程序版本。甚至二者可以同时进行（在多人开发模式下，这是经常出现的情况，后续单元会再详细讨论）。但是在切换到 develop 分支之前，记得先把新分支的状态存入 Git 文档库，然后执行

以下指令：

```
git checkout develop
```

执行这个指令之后，文件夹中的文件会全部变成 develop 分支的版本，然后就可以回到程序开发工具，开始着手修改。是不是很方便呢？在 develop 分支上完成修改之后，可以用同样的方式切换到新分支，继续未完成的工作。

按照这种开发模式，Git 文档库就会变成如图 10-6 所示的架构。但是这种情况会造成一个问题，就是新分支的程序代码和 develop 分支上的程序代码不一致。这种现象已经在前一个单元讨论过，基本上可以使用 rebase 指令，让 develop 分支的修改运用到新分支上。执行 rebase 指令可能发生“冲突”的情况，解决方法请参考前一个单元的说明。

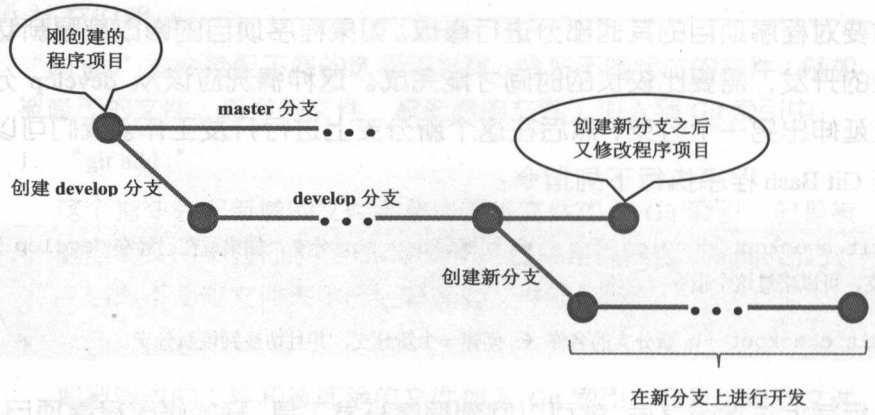


图 10-6　从 develop 分支延伸另一个新分支的开发模式

等到新分支的开发工作完成以后，就可以在 Git Bash 程序执行下列指令，把最后的结果合并到 develop 分支，然后删除新分支。Git 文档库就会回到如图 10-5 所示的架构。

```
git checkout develop
```

```
git merge --no-ff 新分支的名称
git branch -d 新分支的名称
```

另一种情况是最后决定放弃新分支的开发工作，这时候只要在 Git Bash 程序执行下列指令。也就是先切换到 develop 分支，然后强制删除新分支即可。文件夹中的文件会全部恢复到 develop 分支的状态。

```
git checkout develop
git branch -D 新分支的名称
```

在程序项目开发期间，有时候需要回头查看旧版本的程序文件，或是对比不同版本的程序文件，以便找出二者之间的差异。甚至删除 Git 文档库的 commit 节点，回到旧版本的状态。这些相关的 Git 指令和操作技巧，都可以参考前面单元的解说。最后当开发工作大功告成的时候，就可以切换到 Git Bash 程序，执行以下指令。把 develop 分支上的程序项目合并到 master 分支，变成正式版本。Git 文档库就变成如图 10-2 所示的形态。

```
git checkout master
git merge --no-ff develop
```

这个单元我们详细介绍 Git 的使用策略，以及如何将它导入程序项目开发的过程，让读者能够学以致用，实际体验 Git 强大的程序代码管理能力。这个范例主要是以单人开发模式来考虑，但是其实在多人开发模式下，才更能够突显 Git 的优点。本书后续单元会继续带领读者学习 Git 的高级用法。

第三部分　Git 程序的图形操作介面

第 11 课　Git GUI 程序的基本功能

第 12 课　使用 Git GUI 程序创建分支和合并

第 13 课　SmartGit 程序操作介绍

第 14 课　SourceTree 程序操作介绍

第 15 课　TortoiseGit 程序操作介绍



第11课

Git GUI 程序的基本功能

到当前为止，我们一直都是使用 Git Bash 程序来学习 Git 的用法。Git Bash 程序是以输入指令的方式操作 Git 文档库。与图形操作界面的程序相比，这种命令行模式的程序好像比较麻烦。其实不然，Git Bash 程序才是能够完整发挥 Git 最大功能的操作工具。初学者应该从 Git Bash 程序开始学习，才能打下扎实的 Git 指令基础，并且完整了解 Git 文档库的工作原理。等到熟练掌握这些基础知识之后，再来学习图形操作模式，自然就能触类旁通、轻而易举。经过前面单元的学习和演练，相信读者已经具备完整的基础。这个单元我们就来介绍 Git GUI 这个图形界面操作程序的用法。

补充说明

在 Git Bash 程序执行“git gui”指令也可以启动 Git GUI。

Git GUI 是和 Git Bash 一起安装的程序，我们可以在 Windows 操作系统的程序集中的 Git 子文件夹找到它。将它启动之后，就会看到如图 11-1 所示的对话框。对话框中会显示下列 3 个选项：

1. Create New Repository

 单击这个选项之后，会让我们输入一个文件夹路径，程序会在这个指定的文件夹路径创建一个 Git 文档库。这个功能就如同在 Git Bash 程序中执行“git init”指令。

2. Clone Existing Repository

 这个选项用来复制一个已经存在的程序项目和它的 Git 文档库。选择这个选项之后，会让我们输入程序项目的文件夹路径和要复制过去的文件夹路径。

3. Open Existing Repository

 选择这个选项之后，会让我们输入一个文件夹路径，里面是已经创建好的 Git 文档库。Git GUI 程序会切换到这个文件夹，然后开始运行。这项功能就如同在 Git Bash 程序中，使用“cd”指令切换到程序项目的文件夹。Git 程序会自动读取 Git 文档库的状态。

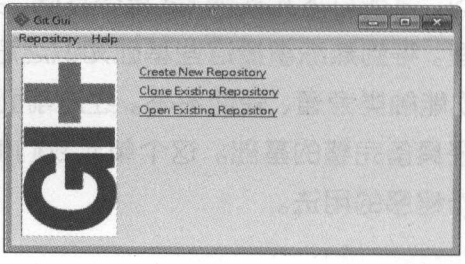


图 11-1　启动 Git GUI 程序后显示的对话框

接下来我们就以第 10 单元的程序项目开发模式为例，介绍如何使用 Git GUI 程序来完成 Git 文档库的操作。

11-1 Git GUI 程序的基本操作

首先必须在程序项目文件夹中创建“.gitignore”文件，排除不需要存入 Git 文档库的文件和文件夹。接下来就可以启动 Git GUI 程序，单击 Create New Repository 选项，输入程序项目的路径。单击 Create 按钮之后，就会显示如图 11-2 所示的屏幕画面。Git GUI 程序的操作界面分成 4 个部分。左上方的区域会显示当前程序项目文件夹内有哪些文件在加入到 Git 文档库之后又被修改了，以及有哪些文件是处于 untracked 状态（也就是没有被加入文档库，而且又不在“.gitignore”文件中）。另外，如果我们从程序项目文件夹中删除文件，该文件也会出现在这个区域，提醒我们要更新这个文件在 Git 文档库的状态。Git GUI 程序左下方的区域是显示当前已经加入 Git 索引的文件。我们可以使用右边的“Stage Changed”按钮，把左上方区域显示的文件加到 Git 索引中。其实 Git GUI 程序左边两个区域显示的信息，就是在 Git Bash 程序中执行“git status”指令所显示的信息。如果文件夹中的文件又被修改了，我们可以单击右边的 Rescan 按钮更新这两个区域的内容。

补充说明

单击“Stage Changed”按钮是把“当前”文件内容登录到 Git 索引。如果之后又修改了文件，这些新修改的内容不会自动加到 Git 索引。我们必须重新单击“Stage Changed”按钮，才会把最新的文件内容登录到 Git 索引。

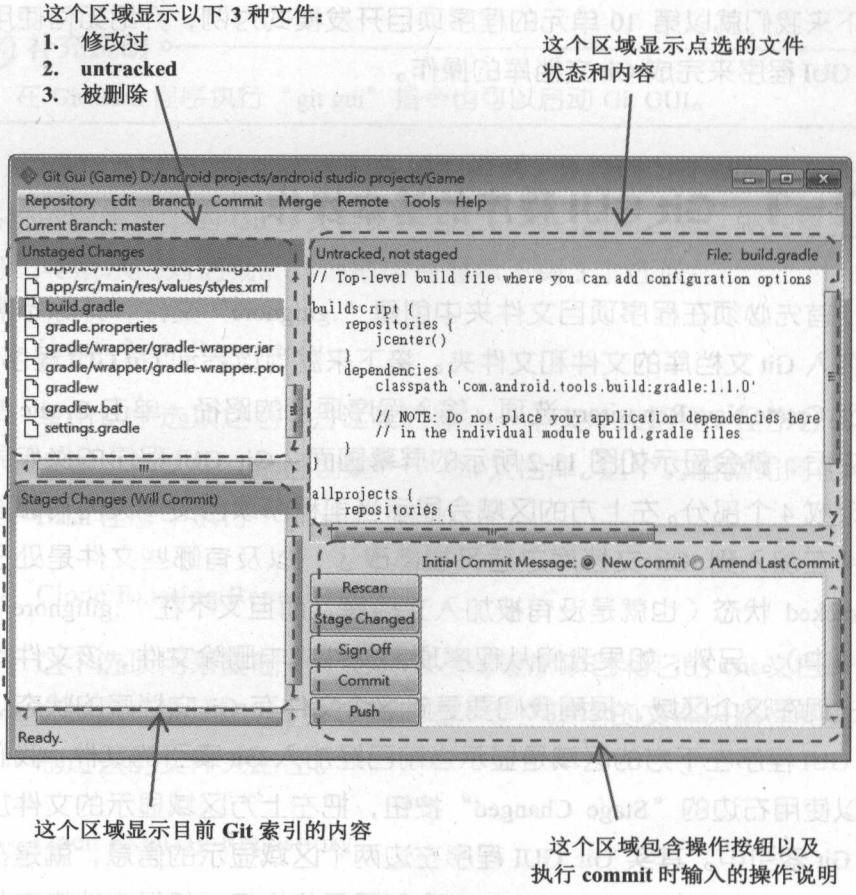


图 11-2　Git GUI 程序的操作画面

如果用鼠标单击左边区域显示的文件名，该文件的相关信息会显示在右上方区域。另外单击“Stage Changed”按钮时，程序会先检查文件夹中是否有 untracked 状态的文件。如果有，就会显示对话框，询问是否要把 untracked 状态的文件登录到 Git 索引。如果选择“是”，程序会把所有 untracked 的文件、被修改的文件和被删除的文件全部登录到 Git 索引。这就等于在 Git Bash 程序中执行“git add -A”指令。如果选择“否”，程序只会把被修改的文件和被删除的文件登录到 Git 索引。这就等于在 Git Bash 程序中执行“git add -u”指令。除了使用“Stage Changed”按钮把文件的状态或是内容登录到 Git 索引之外，也可以单击左边区域的

文件名前面的小图标来实现相同的功能。同样的操作技巧也可以把文件从 Git 索引中删除。如果想要把某些文件加入 Git 索引，可以配合键盘上的 Shift 键或是 Ctrl 键，先选好文件，然后选择主菜单的 Commit > Stage To Commit。如果要从 Git 索引删除某些文件，同样先选好文件，再选择主菜单的 Commit > Unstage From Commit。

把要更新的文件内容或是文件状态加到 Git 索引之后，接下来就是执行 commit。执行 commit 需要输入操作者的资料以及操作说明。操作者的资料可以使用主菜单的 Edit > Options 进行设置，如图 11-3 所示。设置值有两种有效范围，左边的设置只对当前这个程序项目有效，右边的设置则对所有 Git 文档库都有效（提示：关于 Git 配置文件的存储位置和影响范围请参考第 2 单元的说明）。图 11-3 中除了标示用户数据字段以外，另外也标示出几个常用的设置项供读者参考。设置好用户数据之后，接着在 Git GUI 程序右下方区域输入 commit 说明，然后单击 Commit 按钮，就会把 Git 索引的内容存入文档库。

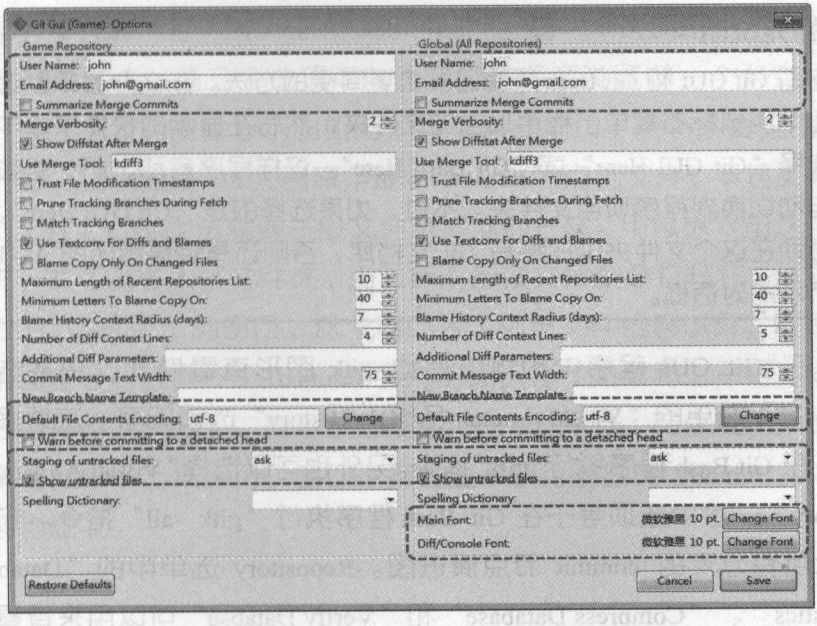


图 11-3　Git GUI 程序的设置画面

11-2 Git GUI 程序和其他程序搭配使用

现在我们已经学会 Git GUI 程序的基本操作。在进一步介绍其他功能之前，先转换一下主题，看看 Git GUI 如何和 Git Bash 程序以及其他程序搭配使用。或许读者会觉得有点奇怪，为什么 Git GUI 程序要和其他程序搭配呢？这是因为 Git GUI 程序只提供常用的功能，高级的 Git 文档库操作还是要通过 Git Bash 程序来执行。使用 Git GUI 的好处是可以比较方便地查看文件夹中的文件变动情况以及 Git 索引的内容。在执行 commit 的时候，也能够很方便地编辑操作说明。当需要执行 Git 指令的时候，可以从主菜单的 Repository > Git Bash 来启动 Git Bash 程序（参考图 11-4）。而且 Git Bash 程序会自动切换到和 Git GUI 程序相同的工作文件夹。当在其中一个程序操作 Git 文档库的时候，另一个程序也会看到最新的状态。也就是说这两个程序的状态是同步的。

补充说明

启动 Git GUI 和 Git Bash 还有另一个更简便的方法。在文件资源管理器中用鼠标右键单击程序项目的文件夹，然后在显示的快捷菜单中选择“Git GUI Here”或“Git Bash Here”。这样程序启动之后，就会自动切换到程序项目文件夹。不过，如果选择的是“Git GUI Here”，必须在这个文件夹中已经有 Git 文档库，否则还是会显示如图 11-1 所示的对话框。

在 Git GUI 程序中也可以启动 gitk 图形查看模式，只要选择 Repository 选单的“Visual (branch 名称)'s History”选项即可。这项操作等于在 Git Bash 程序执行“gitk”指令。另外也可以选择“Visual All Branch History”，这个选项等于在 Git Bash 程序执行“gitk -all”指令。它会显示所有分支的 commit 节点演进图。Repository 选单中的“Database Statistics”、“Compress Database”和“Verify Databse”可以用来查看和

管理 Git 文档库。执行“Compress Database”就等同于在 Git Bash 程序执行“git gc”指令。

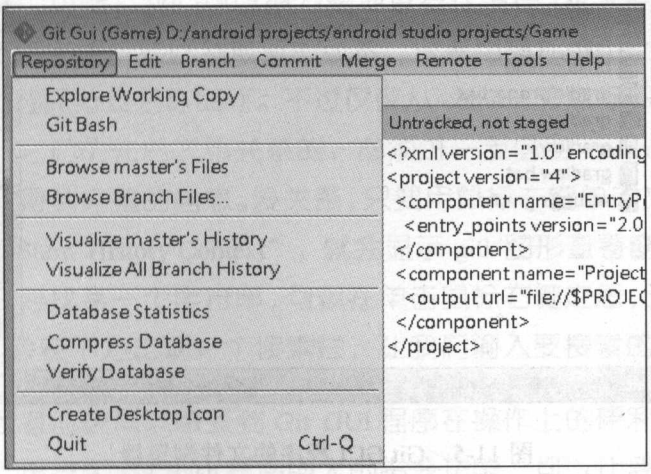


图 11-4　Git GUI 程序的 Repository 选单

Repository 选单中还有两个比较特别的选项，就是“Browse（branch 名称）'s Files”和“Browse Branch Files”。第一个选项是浏览当前这个分支的文件，第二个选项可以让我们选择要浏览哪一个分支的文件。执行这项功能会显示出如图 11-5 所示的窗口。要特别提醒的是，这个窗口显示的是 Git 文档库中最新 commit 节点的文件，不是文件夹中的文件。它的用法就像一般文件浏览窗口一样。用鼠标双击这个文件夹，就可以进入该文件夹。用鼠标双击该文件就会打开它，然后显示如图 11-6 所示的窗口。这个窗口除了列出文件内容以外，在每一行前面都会对应到一个 commit 节点的标识符（如果和上一行相同，就会用垂直线代替），表示该行是在那一个 commit 节点被加入的。如果用鼠标单击某一行，窗口下方会显示该行对应的 commit 节点的详细信息。

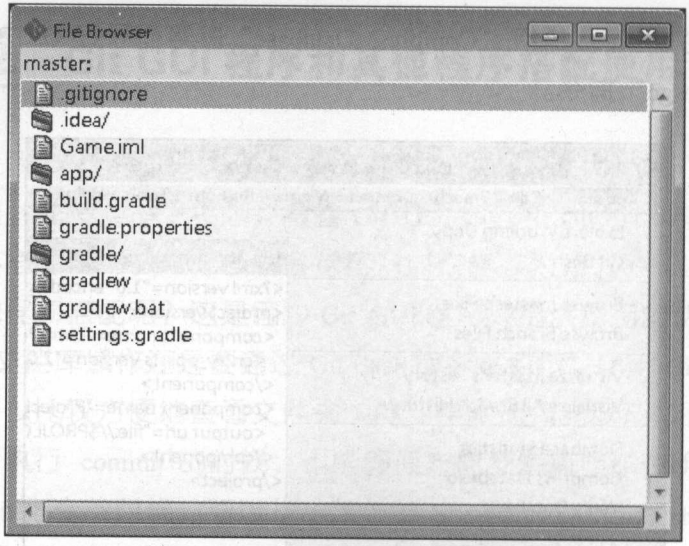


图 11-5　Git GUI 程序的文件浏览器

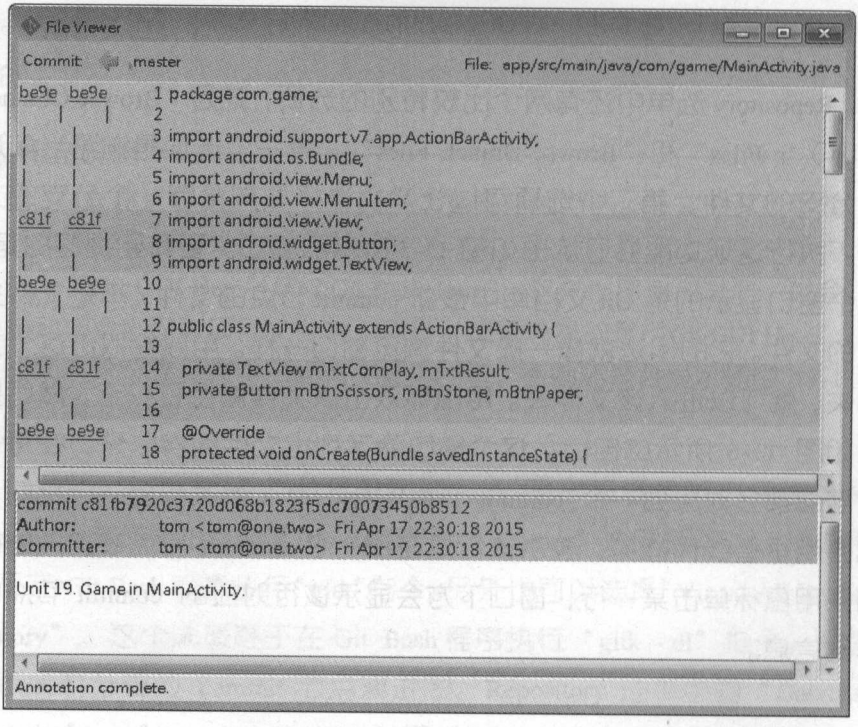


图 11-6　Git GUI 程序的文件窗口

图 11-6 的文件窗口还可以切换不同版本的文件内容，只要用鼠标单击左边的 commit 节点标识符，就会显示该 commit 节点的文件内容。使用这个功能，我们可以很方便地查看存储在 Git 文档库中的每一个文件版本。如果要回到最新的版本，只要用鼠标单击上方状态栏的箭头，再选择分支名称即可。不过仅仅从 commit 节点标识符，我们是无法判断它们的先后次序关系的，必须进一步借助 commit 节点演进图才可以得到完整的信息。没关系，只要用鼠标右键单击文件内容，然后选择“Show History Context”，就会显示 gitk 图形查看窗口。如果要在文件中寻找某一个字符串，可以在单击鼠标右键之后，选择“Find Text”，就会在下方出现一个搜索栏，让我们输入要搜索的文字。

现在读者应该可以感受到 Git GUI 程序在操作上的便利性。使用它的优点，再搭配 Git Bash 程序强大的指令功能，可以让我们更有效率地使用 Git。下一个单元我们要继续介绍如何使用 Git GUI 程序来创建分支和合并。



第12课

使用 Git GUI 程序创建分支和合并

我们在第 10 章详细讨论过使用 Git 的策略。按照建议的程序项目开发流程，分支的创建和合并是必要的步骤，而且这部分的操作也可以使用 Git GUI 程序来完成。Git GUI 程序把分支相关的操作收集在主菜单的 Branch 选项中，如图 12-1 所示。Branch 菜单的第一个选项是 Create，它是用来创建一个新的分支。选择这个选项后，会显示如图 12-2 所示的对话框。在最上方的 Name 字段输入新分支的名称，然后单击右下角的 Create 按钮，就会在 Git 文档库中创建一个新分支，并且自动切换到新分支。主菜单下方的状态栏会显示当前所在的分支，如图 12-3 所示。

补充说明

在创建新分支的对话框最下面有一个“Checkout After Creation”选项。在默认情况下，这个选项会被勾选。因此创建新分支之后，会自动切换到新分支。如果我们把它取消，就不会自动切换到新分支。

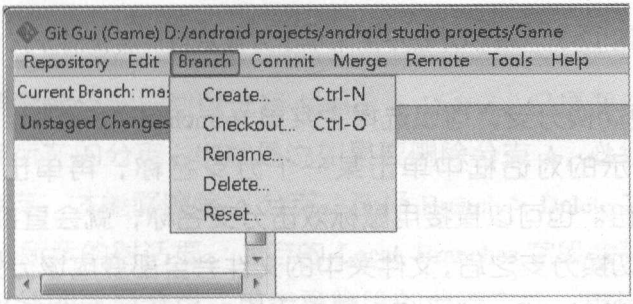


图 12-1　Git GUI 程序的 Branch 菜单

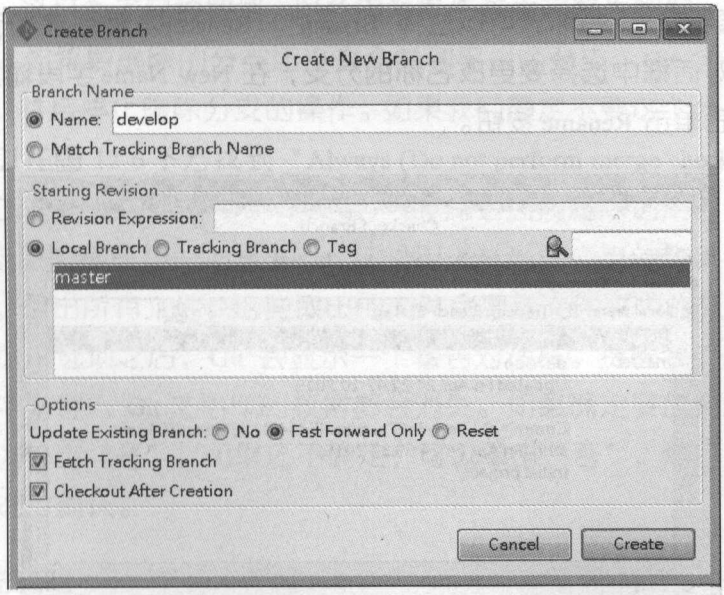


图 12-2　创建分支时显示的对话框

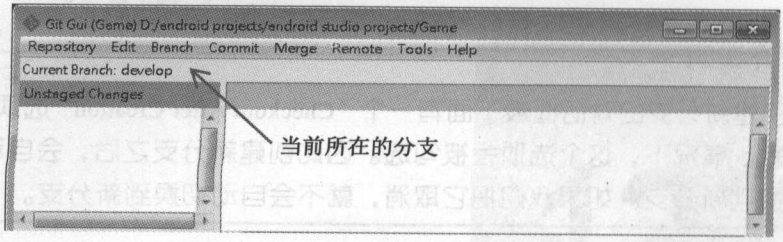


图 12-3　Git GUI 程序主菜单下方显示当前所在的分支

如果要切换分支，可以选择主菜单 Branch > Checkout。然后在如图 12-4 所示的对话框中单击某一个分支名称，再单击右下角的 Checkout 按钮。也可以直接用鼠标双击分支名称，就会直接切换到选定的分支。切换分支之后，文件夹中的文件会全部变成该分支的版本。如果想要知道某一个分支上的 commit 信息，可以把鼠标光标移到该分支上，然后暂停 1 秒钟，就会显示该分支最新的 commit 节点信息。如果要更改分支名称，可以选择 Branch > Rename。然后在如图 12-5 所示的对话框中选择要更改名称的分支，在 New Name 字段输入新名称，最后单击 Rename 按钮。

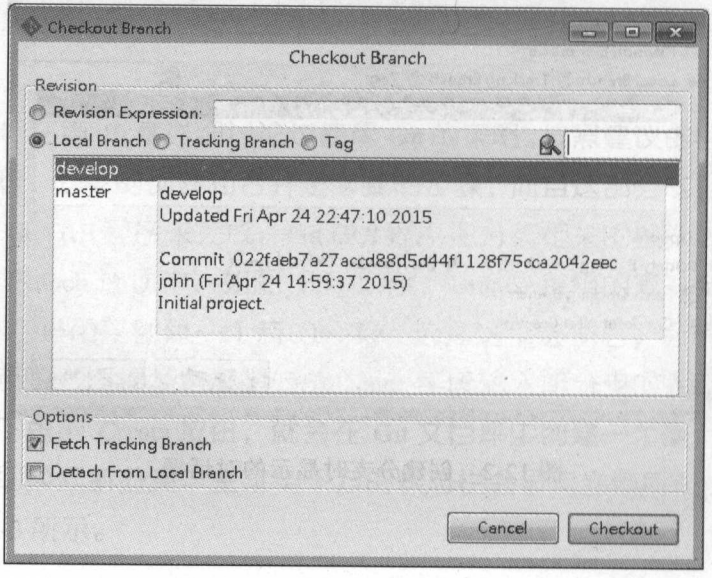


图 12-4　切换分支的对话框

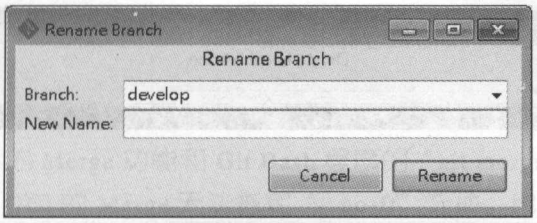


图 12-5　更改分支名称的对话框

如果要删除分支，可以选择 Branch > Delete。但是要注意，不可以删除当前所在的分支。也就是说如果要删除分支 A，必须先切换到 A 以外的分支，才能够删除 A 分支。选择 Branch > Delete 之后，会出现如图 12-6 所示的对话框。上方的 Local Branches 字段会列出当前所在的分支以外的所有分支。单击要删除的分支之后，单击右下方的 Delete 按钮。Git 在删除分支以前，会先检查该分支是否已经合并到其他分支。这是为了保护数据，以免因为操作疏失而遗失数据。在默认情况下，如果要删除的分支还没有合并到另一个分支，Git 会显示警告信息，并且停止删除分支的操作。如果我们确定不要这个分支的数据，可以在图 12-6 下方找到“Always (Do not perform merge checks)”选项。单击它，然后单击 Delete 按钮，Git 就会强制删除选定的分支。Branch 选单中的最后一项是 Reset，它是用来从当前分支的最新 commit 节点中，取出所有文件。这些取出的文件会覆盖文件夹中的文件。因此选择这个选项之后，Git 会显示一个警告对话框。提醒我们，当前文件夹中未存入 Git 文档库的数据都会消失。如果确定要放弃这些数据，可以单击“是”。如果还不确定，可以单击“否”，然后回到主程序再进行确认。

补充说明

执行 Branch > Reset 之前，可以先单击 Rescan 按钮，看看当前文件夹中有哪些文件被修改了，再决定是否要放弃被修改的文件。

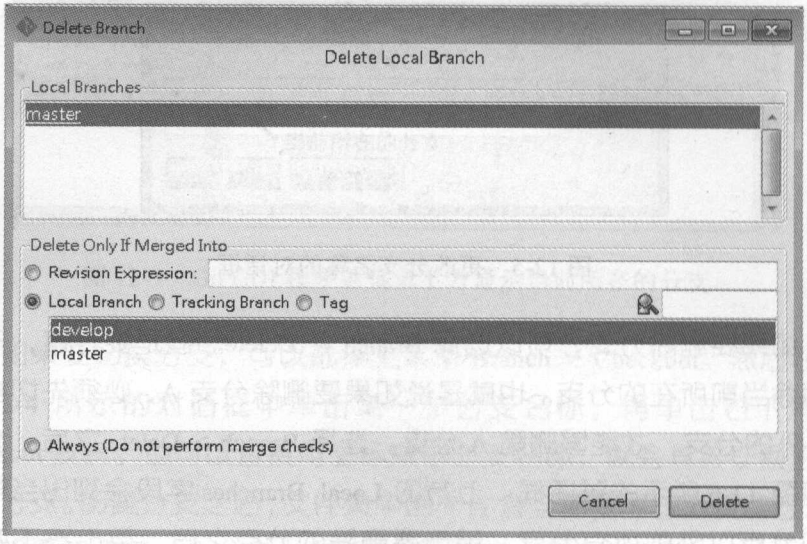


图 12-6　删除分支的对话框

如果要进行分支的合并，可以使用主菜单的 Merge > Local Merge。假设我们要把分支 A 合并到分支 B，首先使用菜单 Branch > Checkout 切换到分支 B。然后选择 Merge > Local Merge，就会出现如图 12-7 所示的对话框。对话框中会列出分支 B 以外的所有分支。从中单击要合并的分支，再单击 Merge 按钮，就会开始进行合并。如果要查看 Git 文档库当前分支的状态，可以单击对话框左边的 Visualize 按钮。它会启动 gitk 图形查看模式。Git GUI 执行合并的过程和 Git Bash 程序的“git merge”指令完全一样，也就是说，同样可能发生“冲突”的情况。如果合并的过程出现“冲突”，会在另一个对话框中显示说明信息。这时候我们有两个选择：一是放弃合并；二是自行解决冲突，然后执行 commit。如果要放弃合并，可以从主菜单中选择 Merge > Abort Merge。文件夹中的文件以及 Git 文档库都会恢复到未合并之前的状态。如果要自行解决冲突，可以选择主菜单的 Repository > Git Bash，启动 Git Bash 程序。然后按照第 8 单元的说明，编辑发生冲突的文件。或者也可以使用 mergetool 来帮忙解决冲突的文件。完成文件的编辑之后，就可以回到 Git GUI 程序。再使用上一个单元介绍的方法，把修改的文件登

录到 Git 索引，最后执行 commit，就正式完成了分支的合并。

补充说明

Git GUI 程序的 Merge 功能和 Git Bash 程序的“git merge”指令的差别在于，Git GUI 的 Merge 无法指定“--no-ff”选项。也就是说，Git GUI 程序的 Merge 功能会自动判断是否使用 fast-forward merge，我们无法强制取消 fast-forward merge。

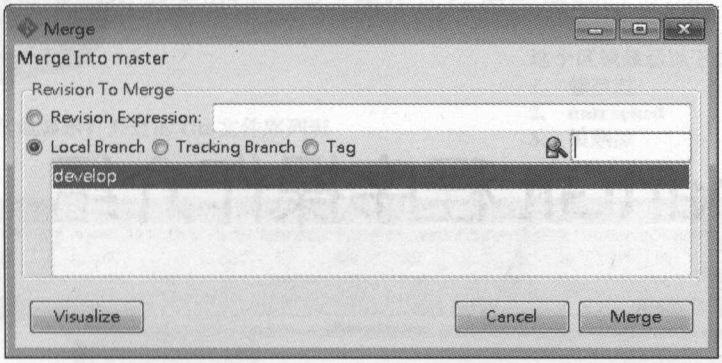


图 12-7　合并分支的对话框



第13课

SmartGit 程序操作介绍

SmartGit 和 Git GUI 程序一样，都是采用图形操作界面。不同的是 SmartGit 程序支持比较完整的 Git 功能，但是也由于功能比较多的缘故，SmartGit 的操作界面比 Git GUI 程序复杂许多。初次使用 SmartGit，常常不知从何下手。没关系，只要了解 Git 基本概念和操作方法，再配合这个单元的介绍，很快就能够学会使用 SmartGit。

首先是下载 SmartGit 并且完成安装。请开启网页浏览器，连接到以下网址，就可以找到 SmartGit 的安装文件：

http://www.syntevo.com/

也可以使用百度或者 Google 搜索页搜索 SmartGit，都可以找到下载网址。SmartGit 提供非商业用途的免费使用模式，因此我们可以顺利地完成安装。启动 SmartGit 程序之后，会看到如图 13-1 所示的屏幕画面。SmartGit 的操作界面包含许多窗口，分别用来显示不同的信息和进行不同的操作。这个单元将用一个实际的程序项目，来示范如何使用 SmartGit 程序来管理程序项目开发时期的各种版本。我们将以第 10 课讨论的 Git 使用策略来进行，整个操作过程将会用到以下功能：

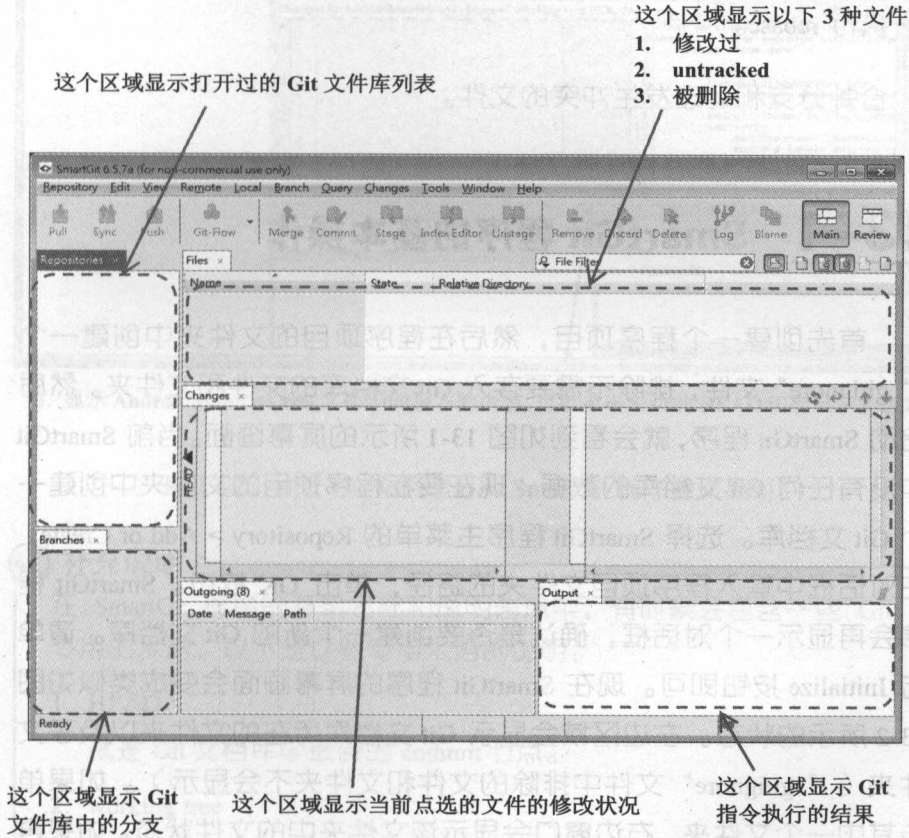


图 13-1　SmartGit 程序的操作界面

1. 创建 Git 文档库。
2. 将文件登录到 Git 索引，以及从 Git 索引删除已经登录的文件。
3. 执行 commit。
4. 查看 Git 文档库的 commit 节点演进图。
5. 创建分支。
6. 执行 rebase。
7. 合并分支和处理发生冲突的文件。

13-1 SmartGit 程序的基本操作

首先创建一个程序项目，然后在程序项目的文件夹中创建一个“.gitignore”文件，排除不需要存入 Git 文档库的文件和文件夹。然后启动 SmartGit 程序，就会看到如图 13-1 所示的屏幕画面。当前 SmartGit 中没有任何 Git 文档库的数据，现在要在程序项目的文件夹中创建一个 Git 文档库。选择 SmartGit 程序主菜单的 Repository > Add or Create，在对话框中输入程序项目文件夹的路径，单击 OK 按钮。SmartGit 程序会再显示一个对话框，确认是否要创建一个新的 Git 文档库。请单击 Initialize 按钮即可。现在 SmartGit 程序的屏幕画面会变成类似如图 13-2 所示的状态。左边区域会显示 Git 文档库所在的文件夹以及子文件夹（“.gitignore”文件中排除的文件和文件夹不会显示）。如果单击其中一个文件夹，右边窗口会显示该文件夹中的文件状态。如果再从中单击一个文件，下方窗口会显示该文件不同版本的差异，包括文件夹中的版本、Git 索引的版本和 Git 文档库中的版本（可以单击窗口左边的 HEAD，或是右边的 Working Tree 来切换显示的版本）。

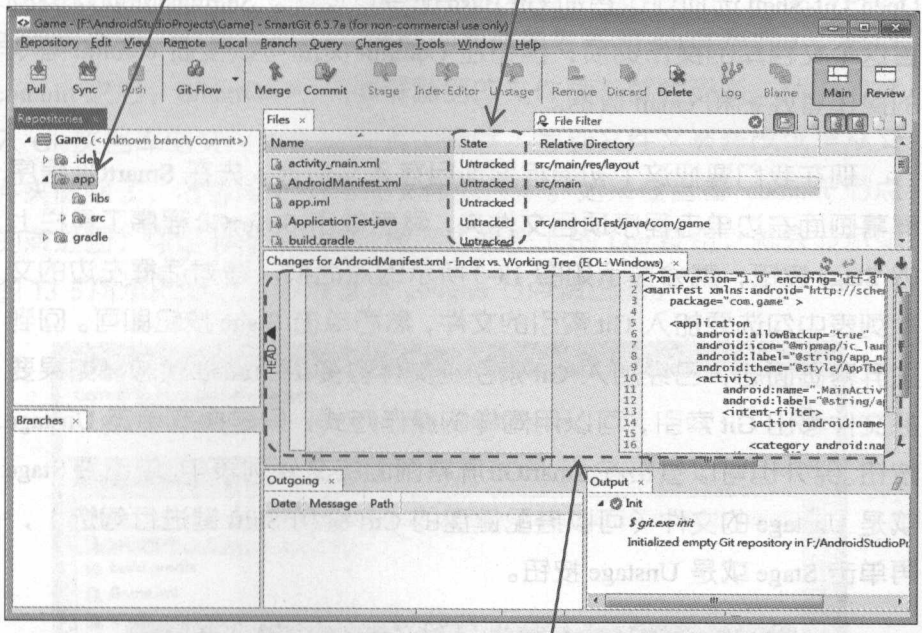


图 13-2　打开 Git 文档库后的 SmartGit 程序操作界面

补充说明

在 SmartGit 程序的菜单名称和说明信息中，有时候会包含一些 Git 专用的术语，以下是这些专用术语的说明：

1. HEAD

 就是 Git 文档库中最新的 commit 节点。

2. working tree

 当前 Git 文档库所在的文件夹，因为文件夹是一个树状结构，所以 Git 将它称为 working tree。

3. index

 就是“Git 索引”，把文件内容加入 Git 索引又称为 stage 或是 cache，把文件内容从索引中删除称为 unstage。

用鼠标右键单击 SmartGit 程序屏幕画面左边的程序项目文件夹（也就是图 13-2 中的 Game 文件夹），会显示一个快捷菜单。里面的 Open Git-Shell 选项可以启动 Git Bash 命令行程序。Settings 选项可以设置这个文档库的操作选项，例如在 Commit 页面可以设置 commit 指令的操作者姓名和 email 数据。

现在我们要把这个初始程序项目存入文档库。先在 SmartGit 程序屏幕画面左边单击程序项目文件夹，然后单击 SmartGit 程序工具栏上的 Stage 按钮，就会显示如图 13-3 所示的对话框。在对话框左边的文件列表中勾选要加入 Git 索引的文件，然后单击 Stage 按钮即可。回到主屏幕画面后，已经加入 Git 索引的文件会变成 Added 状态。如果要把文件移出 Git 索引，可以用同样的操作方式，只是换成单击 Unstage 按钮。另外也可以直接在 SmartGit 屏幕画面的文件列表中，单击要 Stage 或是 Unstage 的文件（可以搭配键盘的 Ctrl 键和 Shift 键进行勾选），再单击 Stage 或是 Unstage 按钮。

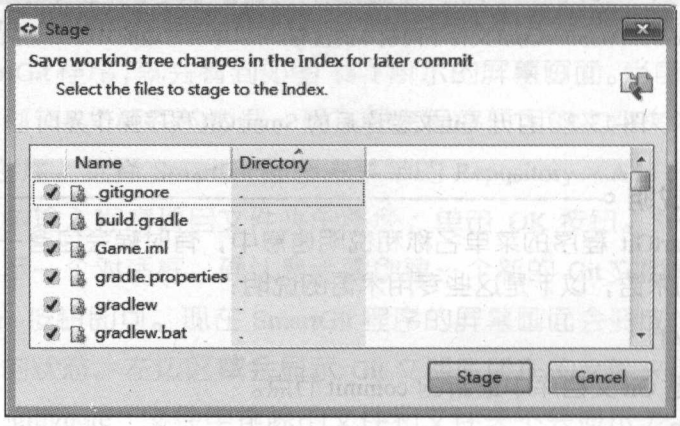


图 13-3　Stage 对话框

把文件加入 Git 索引之后，单击 SmartGit 程序工具栏上的 Commit 按钮，就会显示如图 13-4 所示的对话框。在下方的 Commit Message 字段输入操作说明，然后单击 Commit 按钮即可（提示：必须先按照前面的说明，设置好 commit 指令的操作者姓名和 email 数据）。完成 commit 之后，SmartGit 程序屏幕画面的文件列表就会更新，所有被存入 Git 文档库的文件都会从列表中删除。等到文件又被修改或是从文件夹删除了，才会再度出现在文件列表中。如果要查看 commit 节点的演进图，可以单击 SmartGit 程序工具栏上的 Log 按钮，就会显示如图 13-5 所示的窗口，中间就是 commit 节点演进图。

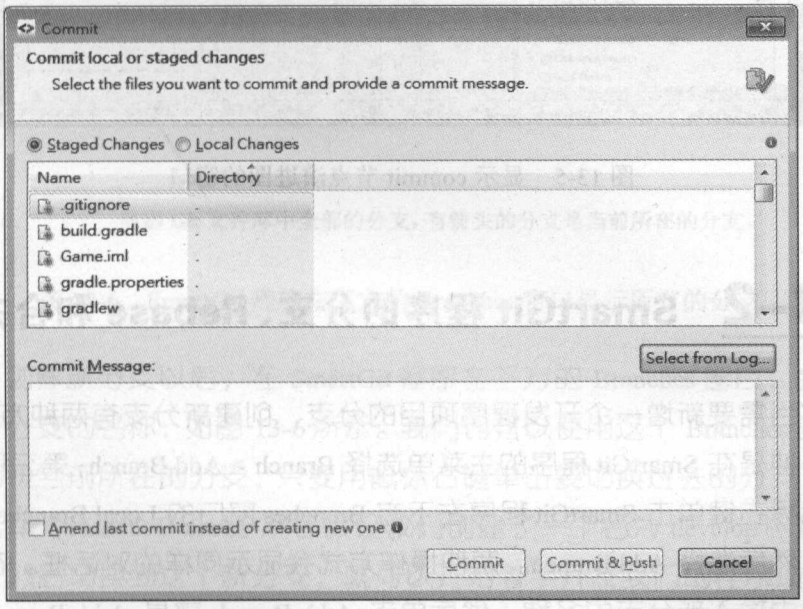


图 13-4　Commit 对话框

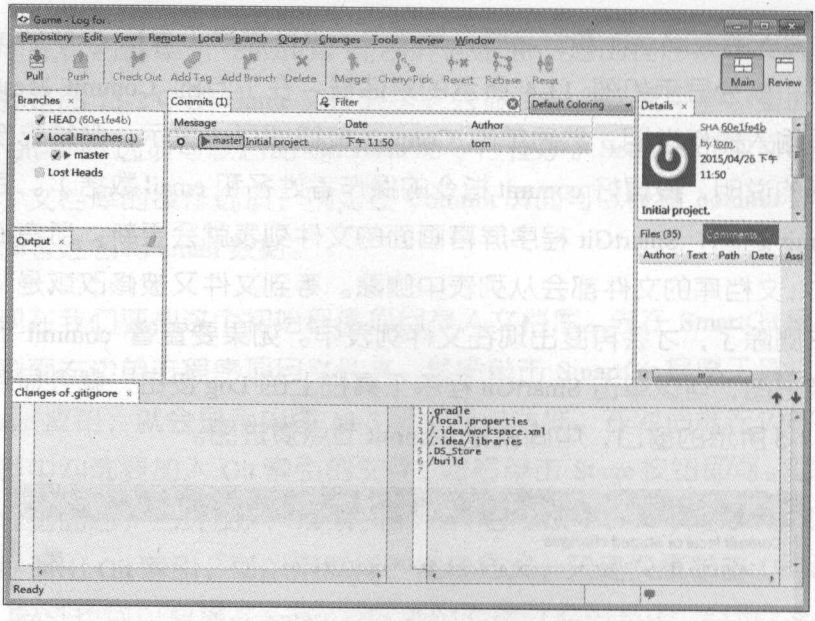
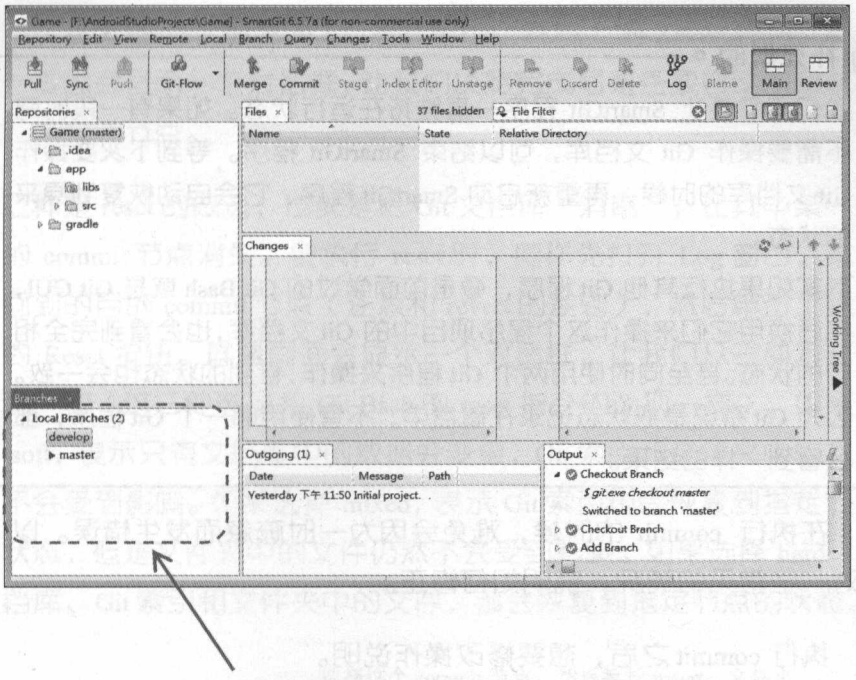


图 13-5　显示 commit 节点演进图的窗口

13–2　SmartGit 程序的分支、Rebase 和合并

当需要新增一个开发程序项目的分支，创建新分支有两种方法：第一种是在 SmartGit 程序的主菜单选择 Branch > Add Branch；第二种是用鼠标右键单击 SmartGit 程序左下方 Branches 窗口的 Local Branches 选项，然后选择 Add Branch。两种操作方式会显示同样的对话框。在对话框中输入新分支的名称；然后单击 Add Branch 或是 Add Branch & Checkout 按钮。第二个按钮除了创建新分支以外，还会自动切换到这个新分支。这个新创建的分支，会从原来分支最新的 commit 节点“长”出来。如果希望可以从其他 commit 节点“长”出新分支，必须换成用以下方式创建分支。首先单击 SmartGit 程序工具栏上的 Log 按钮，然后在如图 13-5 所示的窗口中，单击要“长”出新分支的 commit 节点，再单击工具栏上的 Add Branch 按钮。



显示 Git 文件库中全部的分支，有箭头的分支是当前所在的分支。

图 13-6　SmartGit 程序左下方的 Branches 窗口显示所有的分支

创建新分支以后，在 SmartGit 程序左下方的 Branches 窗口，会显示新分支的名称，如图 13-6 所示。我们也可以使用这个 Branches 窗口来切换当前所在的分支，只要用鼠标右键单击要切换过去的分支，然后选择 Checkout Branch。现在假设我们创建了一个名为 develop 的新分支，然后切换到这个新分支，就可以开始着手开发我们的程序项目。在每一个适当的阶段，可以使用前面小节介绍的方法，把程序项目的状态存入 Git 文档库。在修改程序项目期间，如果切换到 SmartGit 程序，右上方的 Files 窗口会显示有哪些文件被更改、加入或是删除。

补充说明

我们不需要让 SmartGit 程序一直维持在运行状态。如果有一段时间不需要操作 Git 文档库，可以结束 SmartGit 程序。等到下次要操作 Git 文档库的时候，再重新启动 SmartGit 程序，它会自动恢复到原来的状态。

其实如果执行其他 Git 程序，像是前面学过的 Git Bash 或是 Git GUI，然后换用它们来操作这个程序项目中的 Git 文档库，也会看到完全相同的状态。甚至同时使用两个 Git 程序来操作，看到的状态也会一致。因为 Git 系统是把状态记录在磁盘中，不管使用哪一个 Git 程序，都会看到一样的结果。

在执行 commit 的时候，难免会因为一时疏忽而发生错误。以下针对一些常见的情况，说明如何修正。

1. 执行 commit 之后，想要修改操作说明。

 从 SmartGit 程序的主菜单选择 Local > Edit Last Commit Message。在显示的对话框中编辑操作说明，然后单击 OK 按钮。

2. 执行 commit 之后，想要从 Git 文档库中删除这次 commit。

 从 SmartGit 程序的主菜单选择 Local > Undo Last Commit。屏幕画面会显示一个对话框，里面说明将要删除的 commit 节点数据。确认无误后，单击 Undo 按钮。被删除的 commit 节点的文件内容会被送回 Git 索引。

3. 想要让 Git 文档库回到某一个 commit 节点的状态。

 要完成这项工作有两种做法。就如同 Git 指令一样，可以使用 revert 或是 reset 的方式来实现。所谓 revert 是不修改已经存入 Git 文档库的数据，而是把旧数据从指定的 commit 节点取出，这种做法类似于 checkout。要执行 revert 时，单击工具栏上的 Log 按钮，打开 Log 窗口。单击要回到目的 commit 节点的下一个 commit 节点

（参考图 13-7 的说明）。然后单击工具栏上的 Revert 按钮。屏幕画面会显示一个对话框，让我们决定完成后是否要自动创建新的 commit 节点。

第二种是 reset 的做法，也就是把 Git 文档库“消磁”，让其中某一部分的 commit 节点消失。要执行 reset 时，同样先打开 Log 窗口，单击要回到的目的 commit 节点（注意和 revert 的差别），然后单击工具栏上的 Reset 按钮。屏幕画面会显示一个对话框，让我们从三种方式中挑选一种（这三种选项和 Git Bash 的 reset 指令的选项一样）。如果选择 soft，表示只有文档库中的数据会变更，Git 索引和文件夹中的文件都不会受到影响。如果选择 mixed，表示 Git 索引也会恢复到指定节点的状态，但是文件夹中的文件仍然不会受到影响。如果选择 hard，则文档库、Git 索引和文件夹中的文件，都会恢复到指定节点的状态。

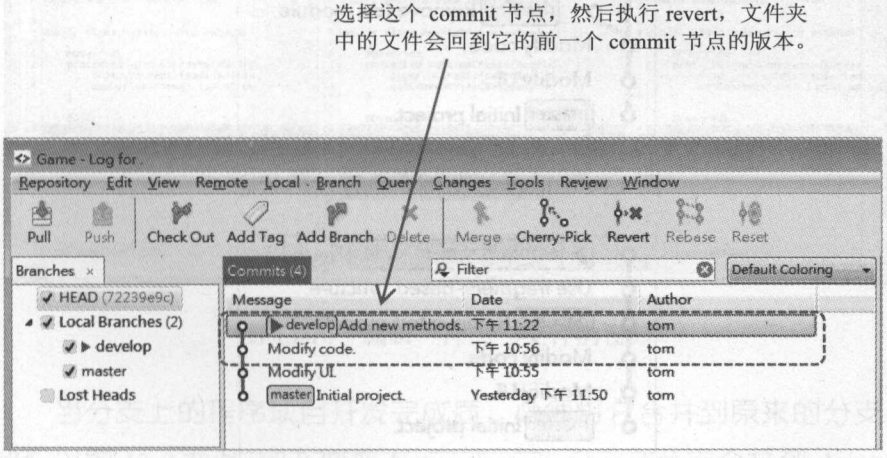


图 13-7　执行 revert 时选择 commit 节点的方式

在第 9 单元已经学过了 Rebase 的原理和用法，那么在 SmartGit 程序如何执行 Rebase？我们用一个例子来说明。请读者参考图 13-8。feature/fragment-based 是从 develop“长”出来的分支。当程序项目在 feature/fragment-based 分支上修改时，develop 分支上的版本也进行了更

新。在这种情况下，我们要把 develop 分支上的修改运用到 feature/fragment-based 分支。这样才能维持基础的一致性。也就是说，要让 feature/fragment-based 分支 Rebase 到 develop 分支上最新的 commit 节点。第一步是切换到 feature/fragment-based 分支，然后单击工具栏上的 Log 按钮，打开 Log 窗口。在 commit 节点演进图中，单击 develop 分支上最新的 commit 节点（表示要 Rebase 到这个节点）。再单击工具栏上的 Rebase 按钮。屏幕画面上会显示一个确认对话框，单击对话框中的“Rebase HEAD to”按钮，就可以完成 Rebase 的操作。结果如图 13-9 所示。如果执行 Rebase 之后，想要恢复到还没 Rebase 前的状态，必须先从主菜单选择 Tools > Open Git-Shell，启动命令行操作模式。然后参考第 9 单元的说明进行操作。

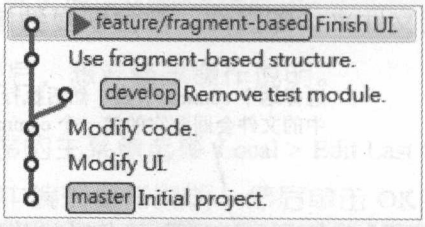


图 13-8　Rebase 范例：执行前的状态

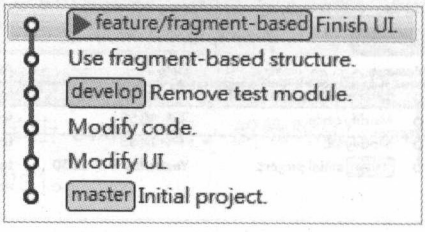


图 13-9 Rebase 范例：执行后的状态

执行 Rebase 的时候，如果两个分支都修改了同一个文件中相同的位置，就会出现发生“冲突”的情况。这时候执行 Rebase 会显示一个对话框，告诉我们执行失败。在这种情况下必须回到 SmartGit 程序的主屏幕画面，在上方的 Files 窗口会显示 Conflicted 状态的文件。用鼠

标双击文件名就会打开文件编辑窗口，如图 13-10 所示。我们必须手动编辑中间窗口的文件，把标示的区域修改为正确的内容。完成之后，单击工具栏的 Save 按钮，然后关闭文件编辑窗口。程序会显示一个对话框，询问是否要把这个文件登录到 Git 索引，以便继续完成 Rebase 的操作，这时单击 Stage 按钮即可。所有 Conflicted 的文件都用相同的方式操作。完成之后，选择 SmartGit 程序主菜单 Branch > Rebase > Continue，就可以完成 Rebase。如果想要放弃这次 Rebase，就可以选择 Branch > Rebase > Abort，文件夹中的文件和 Git 文档库都会恢复到还没执行 Rebase 时的状态。

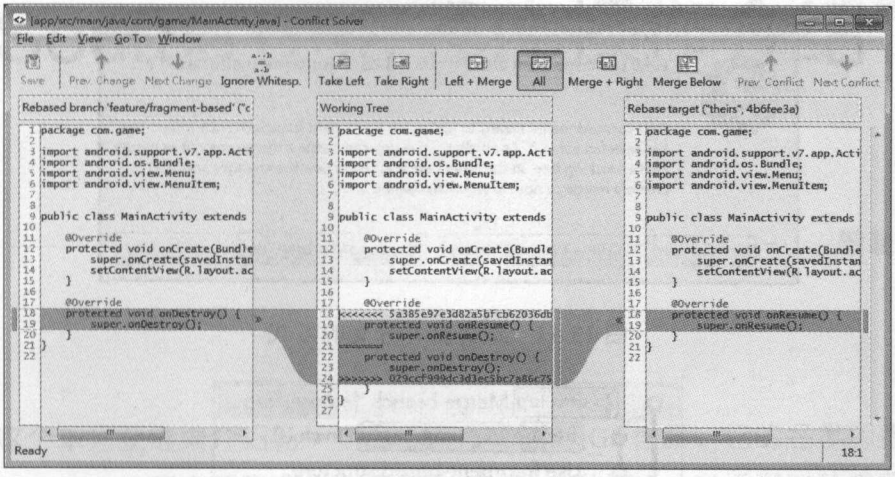


图 13-10　编辑“冲突”文件的窗口

当分支上的程序项目开发完成后，必须将它合并到原来的分支。我们以图 13-9 为例，现在要把 feature/fragment-based 分支合并到 develop 分支。第一步是切换到 develop 分支，然后单击工具栏上的 Log 按钮，打开 Log 窗口。在 commit 节点演进图中，单击 feature/fragment-based 分支上最新的 commit 节点，再单击工具栏上的 Merge 按钮。屏幕画面上会显示如图 13-11 所示的对话框，让我们选择合并的方式。Fast-Forward 和 Create Merge-Commit 这两种方式的差别请参考第 8 单元的说明。Merge to Working Tree 则是把 feature/fragment-based 分支的内容，

合并到当前文件夹内的文件中，但是不会更改 Git 文档库的状态。如果选择 Create Merge-Commit，合并后的结果如图 13-12 所示。执行合并的过程也有可能发生“冲突”。在这种情况下会显示一个对话框，告诉我们执行失败。如果这时候想要放弃合并，可以回到 SmartGit 程序的主屏幕画面，单击工具栏上的 Abort Merge 按钮（就是原来 Merge 那个按钮），文件夹中的文件和 Git 文档库都会恢复到还没执行合并时的状态。如果要继续合并，就要采用类似 Rebase 的做法。也就是先编辑好所有 Conflicted 的文件，并且把这些文件都登录到 Git 索引，最后再单击工具栏上的 Commit 按钮，就可以完成合并。

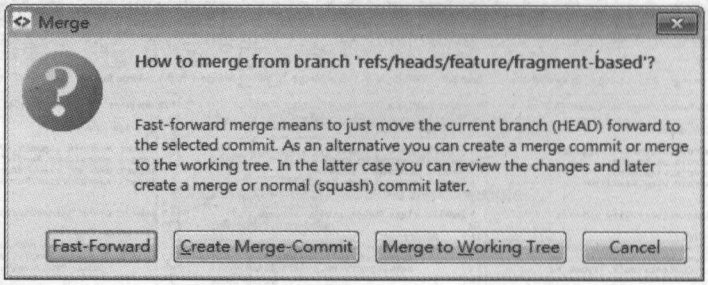


图 13-11　选择合并分支的方式

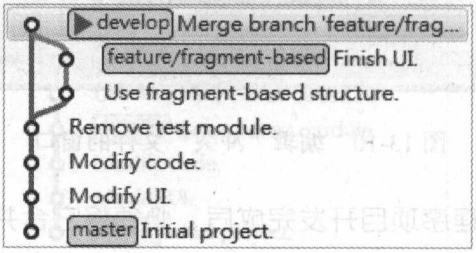


图 13-12　把图 13-9 的 feature/fragment-based 分支合并到 develop 分支后的结果



第14课

SourceTree 程序操作介绍

这个单元要继续介绍另一个图形操作界面的 Git 程序，它的名字叫作 SourceTree。这个程序完全免费，而且功能十分强大。我们可以开启网页浏览器，连接到以下网址下载 SourceTree 程序的安装文件：

https://www.atlassian.com/software/sourcetree/overview

或者使用百度或者 Google 搜索页搜索 SourceTree，也可以找到下载网址。下载之后，执行安装文件，按照屏幕画面的说明操作就可以完成安装。安装完成后，启动 SourceTree 程序，就会看到如图 14-1 所示的屏幕显示画面。由于还没有创建任何 Git 文档库，所以 SourceTree 程序没有显示任何数据。接下来我们会沿用上一个单元的操作流程，

来介绍 SourceTree 的用法。读者也可以借此比较这两个程序的操作方式，看看究竟喜欢哪一个。

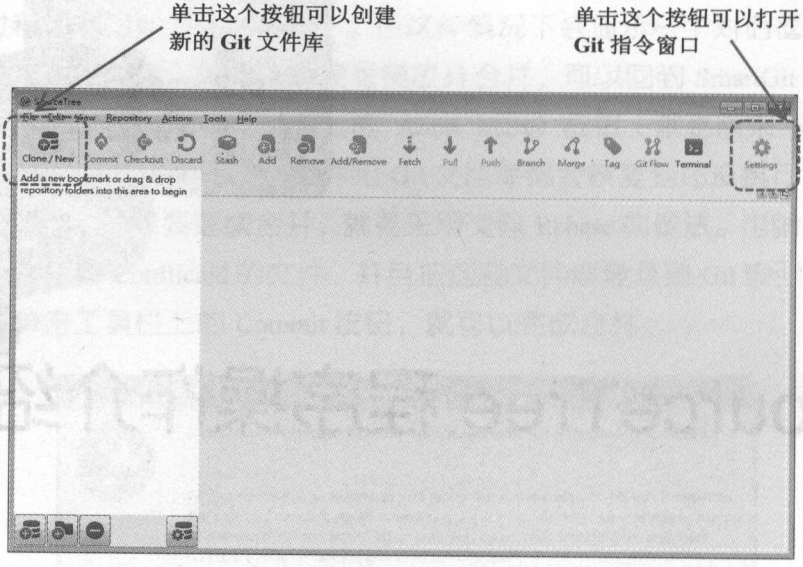


图 14-1　第一次启动 SourceTree 程序时屏幕显示的画面

14–1　SourceTree 程序的基本操作

首先要准备一个程序项目，并且在程序项目的文件夹中创建一个“.gitignore”文件，排除不需要存入 Git 文档库的文件和文件夹。然后启动 SourceTree 程序，单击工具栏最左边的“Clone / New”按钮，就会显示出如图 14-2 所示的屏幕显示画面。我们必须选择 Create New Repository 标签页，再把 Repository Type 设置为 Git，然后在 Destination Path 字段输入程序项目文件夹的路径（可以使用字段右边的按钮，打开文件浏览对话框来设置）。最后单击右下角的 Create 按钮，就会看到如图 14-3 所示的结果。如果我们在 Git 索引的文件列表或是程序项目文件夹的文件列表中单击任何一个文件，该文件的内容就会显示在右边的区域，包括文件中修改的部分也会标示出来。

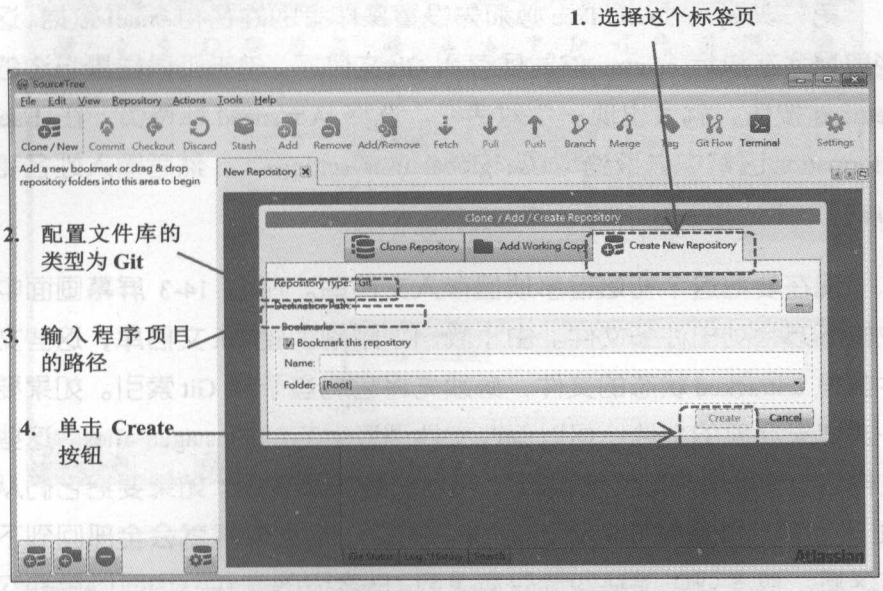


图 14-2　创建 Git 文档库时屏幕显示的画面

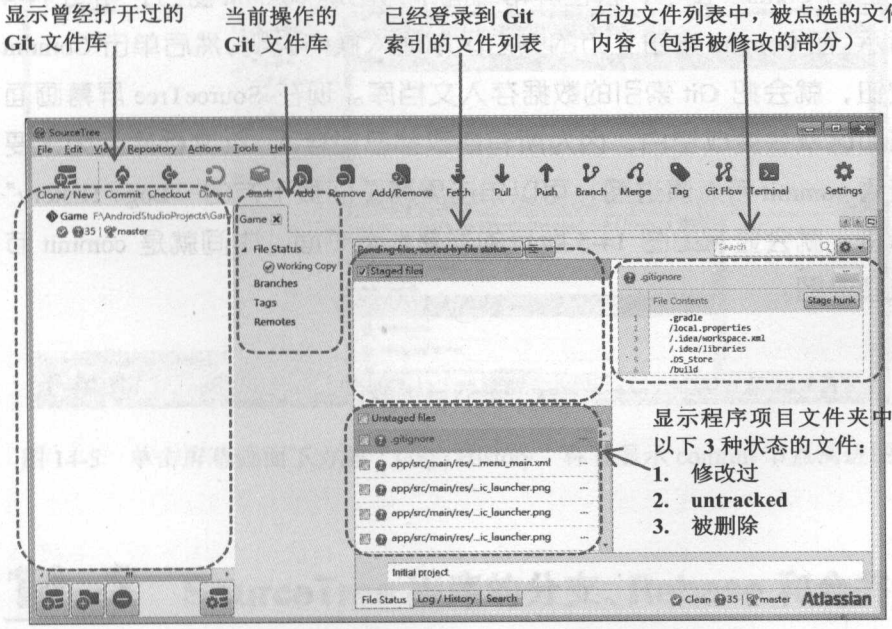


图 14-3　打开 Git 文档库之后屏幕显示的画面

第一次使用 SourceTree 必须先设置操作者的姓名和 email 数据，这些数据会在执行 commit 的时候存入 Git 文档库。单击工具栏最右边的 Settings 按钮，就会出现一个对话框。选择 Advanced 标签页，在 User information 区域中先取消“Use global user settings”，然后输入姓名和 email 数据，再单击 OK 按钮。

现在要把这个初始程序项目存入文档库。在图 14-3 屏幕画面中央偏下方会列出许多文件。由于我们才刚刚创建 Git 文档库，这些文件都是 untracked 状态的文件，必须先将它们登录到 Git 索引。如果要一次登录全部的文件，可以勾选文件列表上方的 Unstaged files。这些文件会立刻出现在上方的区域，表示已经登录完成。如果要把它们从 Git 索引删除，可以勾选上方的 Staged files，这些文件就会全部回到下方区域。或者可以单独勾选要登录到 Git 索引的文件，也可以单独勾选要从 Git 索引删除的文件。把文件登录到 Git 索引之后，单击工具栏上的 Commit 按钮，会在屏幕画面下方出现 commit 窗口，如图 14-4 所示。在 Commit 按钮上方的空白区域输入操作说明，然后单击 Commit 按钮，就会把 Git 索引的数据存入文档库。现在 SourceTree 屏幕画面右边区域会变成空白，因为所有修改都已经存入 Git 文档库。如果要查看 commit 节点演进图，可以单击屏幕画面最下方的“Log / History”标签，就会显示如图 14-5 所示的屏幕显示画面，中间就是 commit 节点演进图。

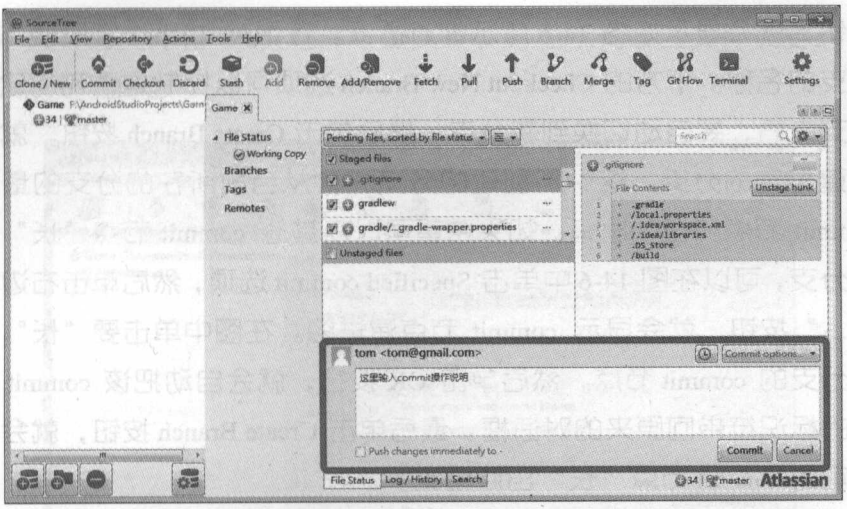


图 14-4　下方是用来执行 commit 的窗口

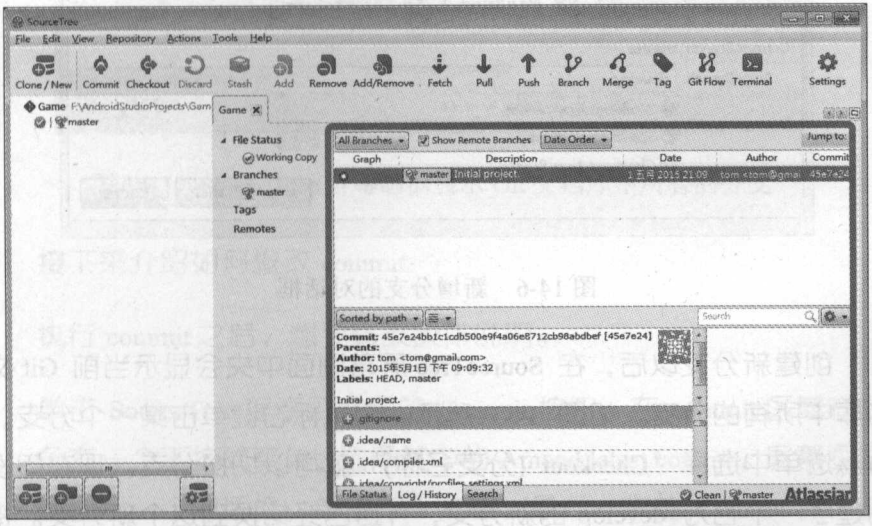


图 14-5　单击屏幕画面下方的“Log / History”标签显示 commit 节点演进图

14-2　SourceTree 程序的分支、Rebase 和合并

现在要新增一个开发程序项目的分支。单击工具栏上的 Branch

按钮，就会显示如图 14-6 所示的对话框。在 New Branch 字段输入新分支的名称。下方的 Checkout New Branch 选项可以控制是否在创建新分支之后，要自动切换到新分支。最后单击 Create Branch 按钮，就会创建指定的分支。这个新创建的分支，会从当前所在的分支的最新 commit 节点“长”出来。如果希望可以从其他 commit 节点“长”出新分支，可以在图 14-6 中单击 Specified commit 选项，然后单击右边的“...”按钮，就会显示 commit 节点演进图。在图中单击要“长”出新分支的 commit 节点，然后单击 OK 按钮，就会自动把该 commit 节点的标识符带回原来的对话框。最后单击 Create Branch 按钮，就会从指定的 commit 节点“长”出新分支。

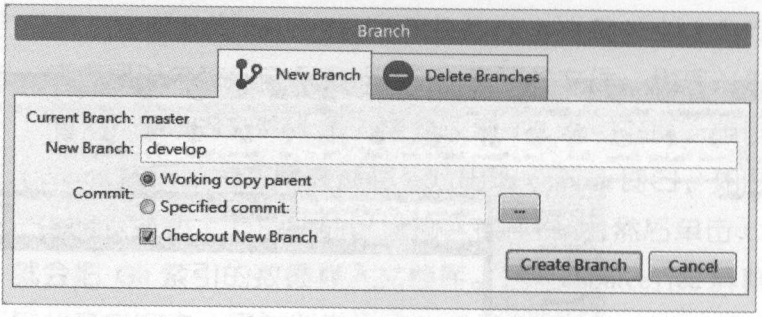


图 14-6　新增分支的对话框

创建新分支以后，在 SourceTree 屏幕画面中央会显示当前 Git 文档库中所有的分支，如图 14-7 所示。用鼠标右键单击某一个分支，再从选单中选择“Checkout (分支名称)”就可以切换分支。现在假设创建了一个名为 develop 的新分支，并且已经切换到这个新分支。接下来可以开始着手开发我们的程序项目。在每一个阶段中，使用前面介绍的方法，把程序项目的状态存入 Git 文档库。在修改程序项目期间，如果切换到 SourceTree 程序，屏幕画面上会显示有哪些文件被更改、加入或是删除。在开发程序项目期间，不需要让 SourceTree 程序一直维持在运行的状态。我们可以随时将它关闭。等到下次要操作 Git 文档库的时候，再启动它。SourceTree 程序启动以后，会自动对比

文件夹中的文件和 Git 文档库中的文件，并显示有哪些文件被更改、加入或是删除。

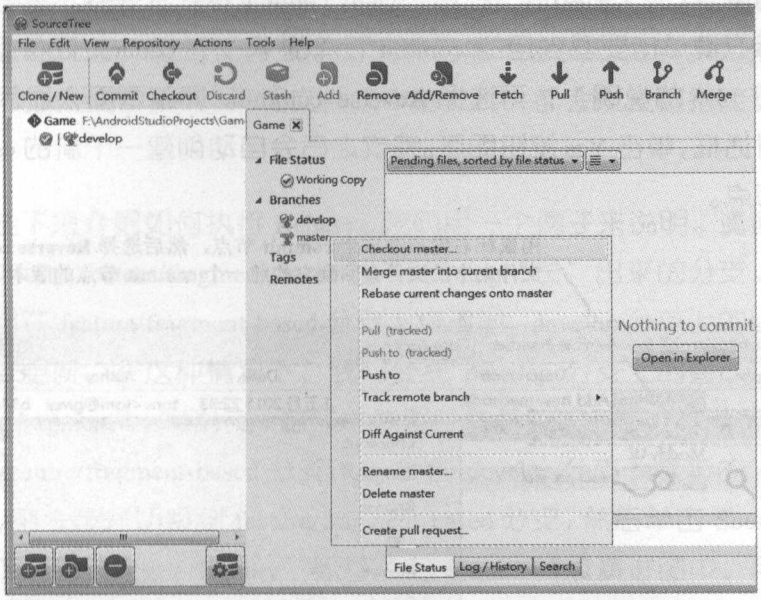


图 14-7　SourceTree 屏幕画面显示 Git 文档库中所有的分支

接下来介绍如何修改 commit：

1. 执行 commit 之后，想要修改操作说明。

 单击 SourceTree 程序工具栏的 Commit 按钮。在 commit 区域右上角有一个下拉式菜单，选择其中的 Amend latest commit。屏幕画面会显示一个对话框，单击 Yes 按钮，最后一次的 Commit 操作说明会出现在 commit 区域中。修改操作说明后，单击 Commit 按钮即可。

2. 让 Git 文档库回到某一个 commit 节点的状态。

 要完成这项工作有两种做法。就如同 Git 指令一样，可以使用 revert 或是 reset 的方式来实现。所谓 revert，就是不修改已经存入 Git

文档库的数据，而是把旧数据从指定的 commit 节点取出，这种做法类似于 checkout。要执行 revert 时，先单击 SourceTree 程序下方的“Log / History”标签，显示 commit 节点演进图。然后用鼠标右键单击要回到目的 commit 节点的下一个 commit 节点（参考图 14-8 的说明），再选择 Reverse commit。屏幕画面会显示一个对话框，单击 Yes 按钮即可。完成之后会自动创建一个新的 commit 节点。

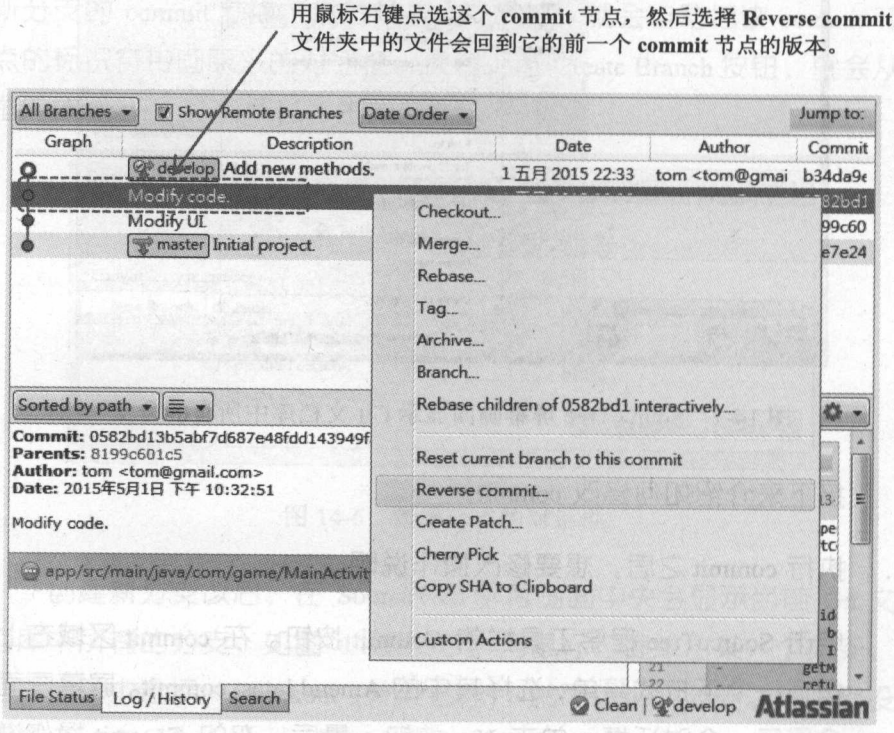


图 14-8　执行 revert 时选择 commit 节点的方式

第二种是 reset 的做法，也就是把 Git 文档库“消磁”，让其中某一部分的 commit 节点消失。要执行 reset 时，同样先单击 SourceTree 程序下方的“Log / History”标签，然后用鼠标右键单击要回到的目的 commit 节点（注意和 revert 的差别），然后选择“Reset current branch to this commit”。屏幕画面会显示一个对话框，让我们从 soft、mixed、hard3

种方式中挑选一种（这 3 种选项和 Git Bash 的 reset 指令的选项一样）。如果选择 soft，表示只有文档库中的数据会变更，Git 索引和文件夹中的文件都不会受到影响。如果选择 mixed，表示 Git 索引也会恢复到指定节点的状态，但是文件夹中的文件仍然不会受到影响。如果选择 hard，则文档库、Git 索引和文件夹中的文件，都会恢复到指定节点的状态。

接下来介绍如何执行 Rebase。我们用一个例子来说明。请读者参考图 14-9。feature/fragment-based 是从 develop“长”出来的分支。当程序项目在 feature/fragment-based 分支上修改时，develop 分支上的版本也进行了更新。在这种情况下，我们要把 develop 分支上的修改运用到 feature/fragment-based 分支。这样才能够维持基础的一致性。也就是说，要让 feature/fragment-based 分支 Rebase 到 develop 分支上最新的 commit 节点。第一步是切换到 feature/fragment-based 分支，然后单击 SourceTree 程序下方的“Log / History”标签。在 commit 节点演进图中，用鼠标右键单击 develop 分支上最新的 commit 节点（表示要 Rebase 到这个节点），再从快捷菜单中选择 Rebase。屏幕画面上会显示一个确认对话框，单击对话框中的 OK 按钮，就可以完成 Rebase。结果如图 14-10 所示。如果执行 Rebase 之后，想要恢复到还没 Rebase 前的状态，必须先从工具栏选择 Terminal，启动 Git 命令行操作模式，然后参考第 9 单元的说明进行操作。

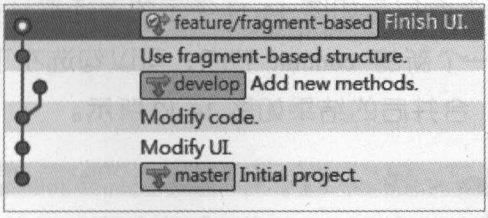


图 14-9　Rebase 范例：执行前的状态

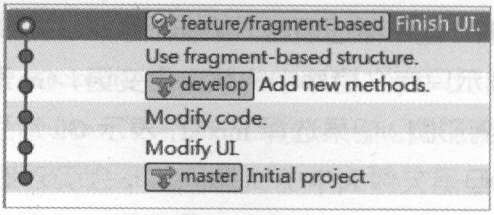


图 14-10　Rebase 范例：执行后的状态

执行 Rebase 的时候，如果两个分支都修改了同一个文件中相同的位置，就会出现发生“冲突”的情况。这时候执行 Rebase 会显示一个对话框，告诉我们发生冲突了。在这种情况下，必须回到 SourceTree 程序的主屏幕画面，在 Unstaged files 窗口会显示 Conflicted 状态的文件。我们必须回到程序项目开发工具，修改所有发生冲突的文件。完成之后，回到 SourceTree 程序，把这些文件登录到 Git 索引。然后选择主菜单 Actions > Continue Rebase。如果想要放弃这次 Rebase，可以选择 SourceTree 程序的主菜单 Actions > About Rebase，文件夹中的文件和 Git 文档库都会恢复到还没执行 Rebase 时的状态。

当分支上的程序项目开发完成后，必须将它合并到原来的分支。以图 14-10 为例，现在要把 feature/fragment-based 分支合并到 develop 分支。第一步是切换到 develop 分支，然后单击 SourceTree 程序下方的“Log / History “标签。在 commit 节点演进图中，用鼠标右键单击 feature/fragment-based 分支上最新的 commit 节点，从快捷菜单选择 Merge。屏幕画面上会显示如图 14-11 所示的对话框。如果希望 Merge 之后一定要创建一个新的 commit 节点，可以勾选左下方的选项，然后单击 OK 按钮。合并后的结果如图 14-12 所示。

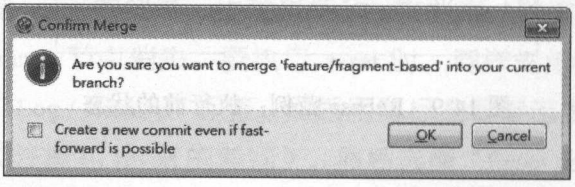


图 14-11　执行合并时显示的对话框

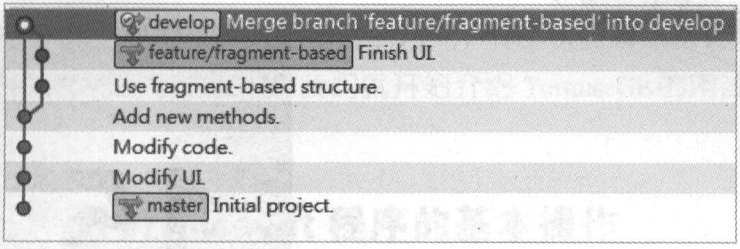


图 14-12　把图 14-10 的 feature/fragment-based 分支合并到 develop 分支后的结果

执行合并的过程也有可能发生“冲突“。在这种情况下会显示一个对话框，告诉我们发生冲突了。如果这时候想要放弃合并，必须从工具栏选择 Terminal，启动 Git 命令行操作模式，然后执行“git merge --abort”指令。Git 文档库和文件夹中的文件都会恢复到未执行合并前的状态。如果要继续合并，就要采用类似 Rebase 的做法。也就是先编辑好所有 Conflicted 的文件，并且把这些文件都登录到 Git 索引，最后再单击工具栏上的 Commit 按钮，就可以完成合并。



第15课

TortoiseGit 程序操作介绍

TortoiseGit 同样是一个图形操作界面的 Git 程序，而且知名度在同类型的程序中算是数一数二的。它是一个开放源码的程序项目，因此完全免费使用。使用百度或者 Google 搜索 TortoiseGit，就可以找到它的官方网址。它提供了 32 位和 64 位的版本，必须按照计算机安装的 Windows 版本，下载合适的安装文件。下载之后，执行安装文件，然后按照屏幕画面的说明操作，就可以顺利完成安装。

TortoiseGit 的操作方式和前面介绍的 Git 程序有些不同。之前介绍的 Git 程序都有一个主屏幕画面，在程序屏幕画面上方有菜单和工具栏，可以进行各种 Git 指令的操作。但是 TortoiseGit 是由许多小工具程序集结而成。这些工具程序没有菜单和工具栏，必须借助屏幕画面

上的按钮，或是使用单击鼠标右键的方式，调出快捷菜单再选择选项进行 Git 指令的操作。接下来我们就开始介绍 TortoiseGit 的用法。

15–1 TortoiseGit 程序的基本操作

假设已经创建好一个程序项目，并且在程序项目的文件夹中创建了一个“.gitignore”文件，排除不需要存入 Git 文档库的文件和文件夹。现在我们要用 TortoiseGit，在这个程序项目的文件夹中创建 Git 文档库。首先启动 Windows 文件资源管理器，用鼠标右键单击这个程序项目的文件夹，就会显示出如图 15-1 所示的快捷菜单。选取其中的“Git Create repository here...”选项，就会显示出一个对话框。不要勾选其中的“Make it Bare (No working directories)”选项，单击 OK 按钮，屏幕画面就显示出另一个对话框，直接单击 OK 按钮，就会在文件夹中创建 Git 文档库。

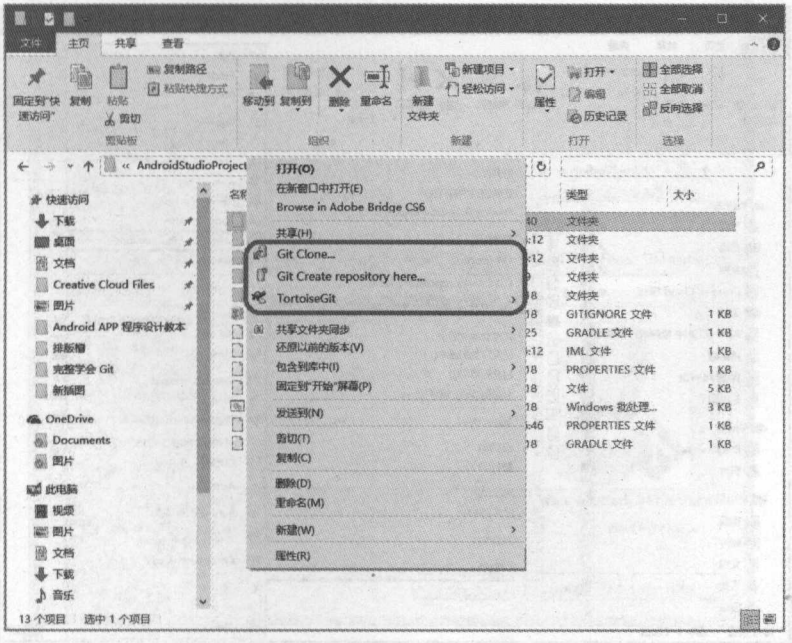


图 15-1 用鼠标右键单击程序项目文件夹显示的快捷菜单

现在我们要把文件夹中的文件存入文档库。用同样的方式调出图 15-1 所示的快捷菜单，选择“TortoiseGit”选项，就会出现第二层的菜单，如图 15-2 所示。单击其中的“Check for modifications”，屏幕画面会出现如图 15-3 所示的对话框。这个对话框称为 Working Tree 对话框，它会显示当前文件夹中新增了哪些文件以及哪些文件被修改了。单击下方的 Commit 按钮，就会显示如图 15-4 所示的 Commit 对话框。对话框下方的文件列表可以让我们勾选想要存入文档库的文件。或者单击文件列表上方的 All、None、Unversioned...（提示：Unversioned 表示新增的文件，也就是 untracked 状态的文件）。完成文件的勾选之后，在上方的 Message 字段输入操作说明，最后单击 OK 按钮。如果执行成功，屏幕画面会显示 Success。如果执行失败，屏幕画面会显示“git did not exit cleanly”。最常见的原因是没有设置操作者姓名和电子邮件。如果出现这种情况，请参考“补充说明”，完成操作者姓名和电子邮件的设置，然后再执行一次。

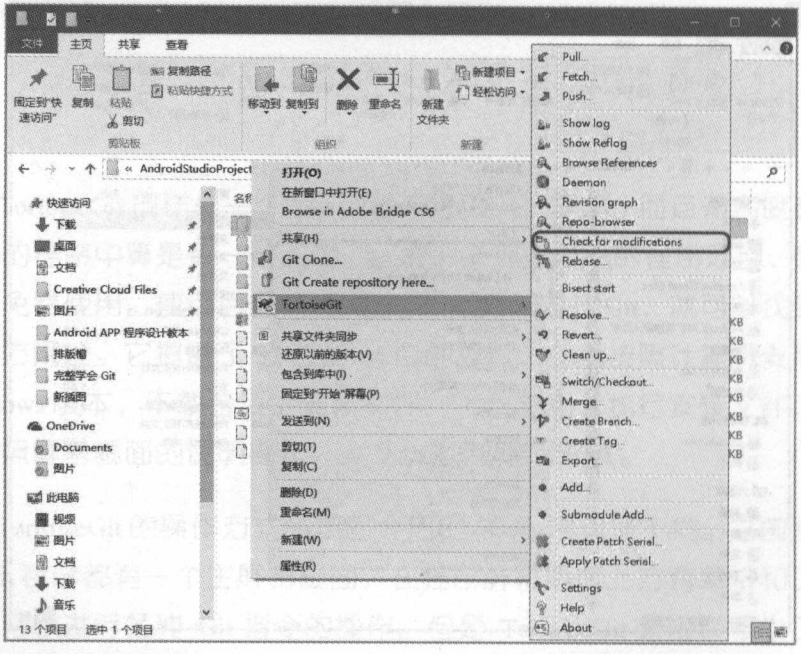


图 15-2　选择快捷菜单的“TortoiseGit”选项出现第二层菜单

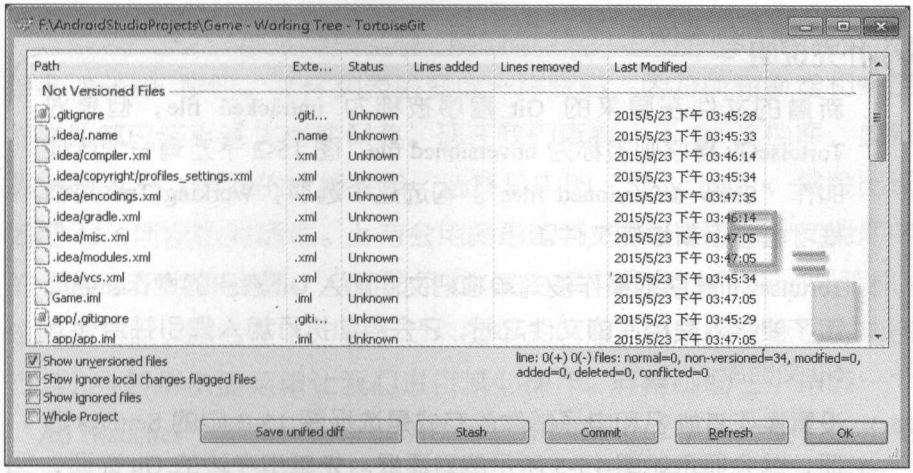


图 15-3　Working Tree 对话框显示新增和修改的文件

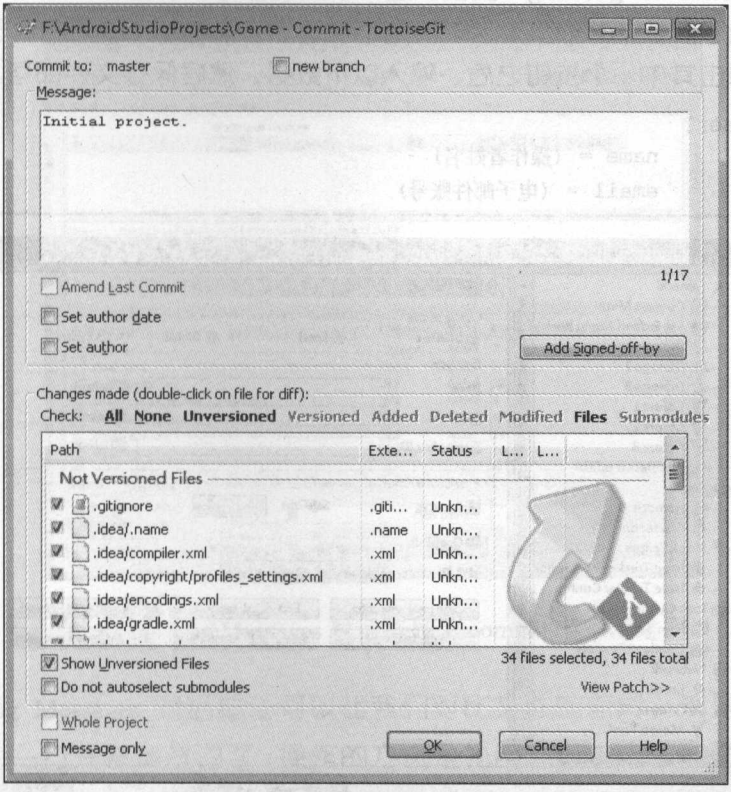


图 15-4　Commit 对话框的操作界面

补充说明

1. 新增的文件在原来的 Git 程序被称为 untracked file，但是对 TortoiseGit 程序则被称为 unversioned file。图 15-3 下方有一个选项叫作“Show unversioned files”。勾选这个选项，Working Tree 对话框才会显示新增的文件。
2. TortoiseGit 程序的操作没有单独把文件加入 Git 索引的操作。当选定了要存入文档库的文件之后，它会自动执行加入索引并且存入 Git 文档库。
3. 设置操作者姓名和电子邮件的方式是选择图 15-2 中的 Settings 选项，就会显示如图 15-5 所示的对话框。先单击左边的 Git 选项，右边会出现“Edit local .git/config”、“Edit global .gitconfig”、“Edit systemwide gitconfig”按钮。这 3 个按钮都可以设置操作者姓名和电子邮件，但是有效范围并不相同（请读者参考第 2 单元的说明）。单击其中一个按钮之后，输入以下数据，然后保存文件即可。

```
[user]
        name = (操作者姓名)
        email = (电子邮件账号)
```

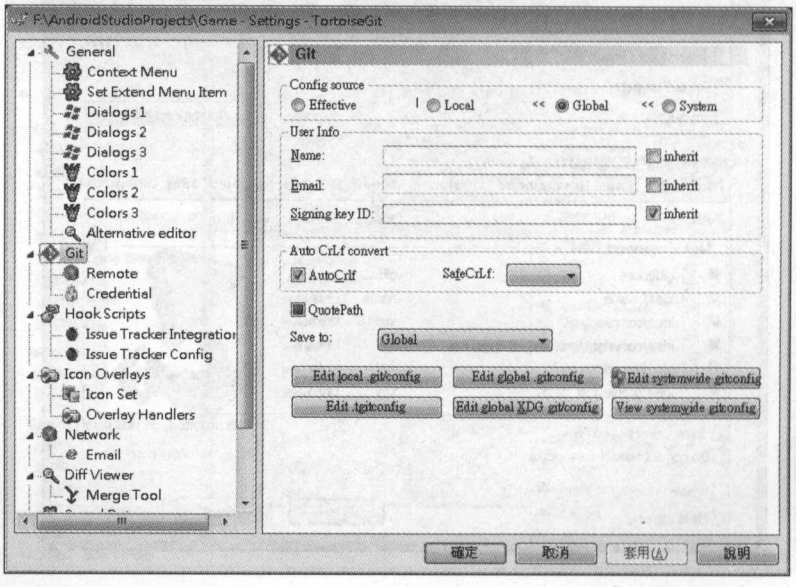


图 15-5　设置操作者姓名和电子邮件

把文件存入 Git 文档库之后，回到图 15-3 的 Working Tree 对话框，单击 Refresh 按钮。文件列表会变成一片空白，因为所有新增和修改的文件都已经送进文档库保存。现在我们查看一下 Git 文档库，重新调出如图 15-2 所示的快捷菜单，选择其中的“Show log”，就会出现如图 15-6 所示的对话框。上方会用图形的方式画出当前 Git 文档库中的 commit 节点演进图。当前显示 Git 文档库中只有一个 commit 节点。单击该节点，就会在下方显示节点的相关信息。如果用鼠标右键单击节点，会显示快捷菜单让我们进行其他操作。屏幕画面左下角有一个“All Branches”选项。它可以切换只显示当前工作中的分支，还是要显示所有分支。如果又执行了新的 commit，可以单击 Refresh 按钮，显示最新的 Git 文档库的状态。

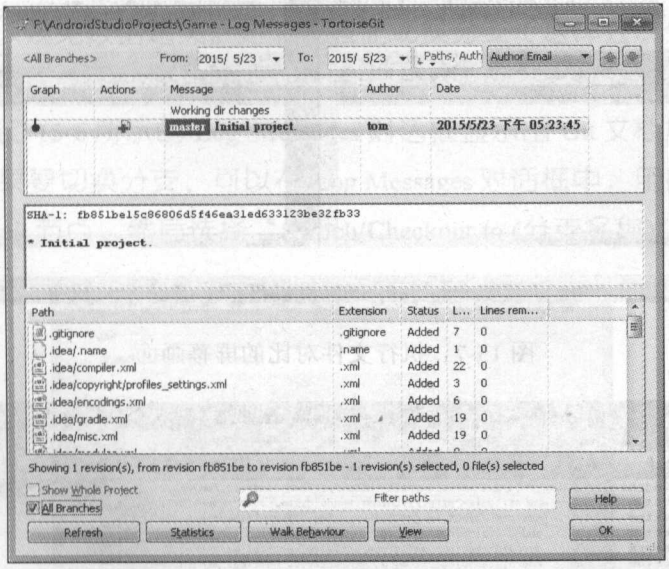


图 15-6　Log Messages 对话框显示 commit 节点演进图

Log Messages 对话框还可以让我们对比文件的差异，或是查看文件内每一行是被谁修改了。操作的方法是先用鼠标单击演进图中的某一个 commit 节点，然后在下方的文件列表中，用鼠标右键单击要查看的文件。然后从快捷菜单中选择 Compare 相关的操作选项，例如

Compare with base，就会显示出如图 15-7 所示的屏幕画面。右边是当前选定的 commit 节点的版本，左边是前一个版本。两个版本的差异都会被标示出来。如果从快捷菜单中选择 Blame，则会显示如图 15-8 所示的屏幕画面。最左边的字段显示每一行是在哪一个 commit 节点修改以及操作者姓名。单击某一个 commit 节点编号，它所对应的文件内容会被标示出来。下方的窗口还会显示和这个文件相关的 commit 节点的演进图。

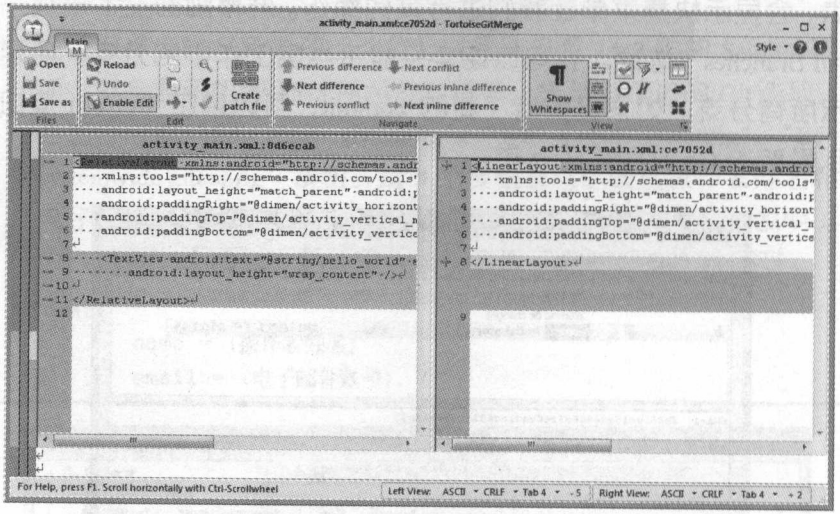


图 15-7　执行文件对比的屏幕画面

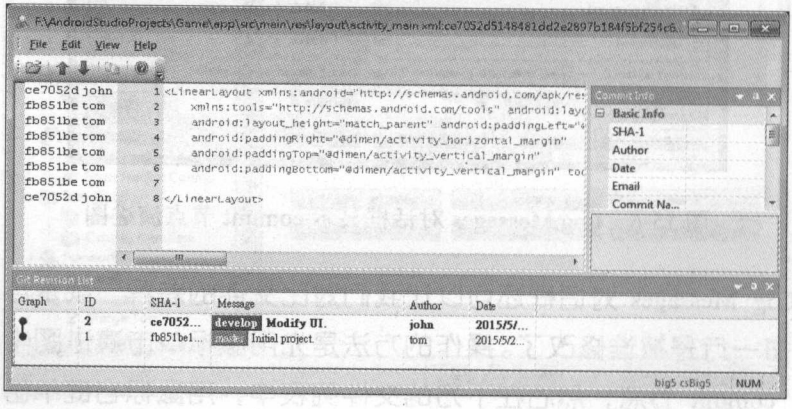


图 15-8　查看文件内容

15-2 TortoiseGit 程序的分支、Rebase 和合并

现在要新增一个开发程序项目的分支。有两种方式可以完成这项操作。第一种是从如图 15-2 所示的快捷菜单选择“Create Branch...”，第二种是在如图 15-6 所示的 Log Messages 对话框中，用鼠标右键单击 commit 节点，然后选择“Create Branch at this version...”。这两种方法都会显示出如图 15-9 所示的对话框。在最上面的字段输入新分支的名称。在 Based On 字段选择 HEAD，表示要从当前所在的分支的最新 commit 节点“长”出新分支。下方可以勾选 Switch to new branch，这样在创建新分支之后，会自动切换到新分支。如果当前所在的分支中，还有修改的文件尚未存入文档库，Git 会显示错误信息，并且停止切换分支，以避免数据遗失。如果要强制切换分支（这样会遗失修改的数据），可以勾选 Force 选项。最后单击 OK 按钮。创建分支之后，可以到如图 15-6 所示的 Log Messages 对话框显示出 Git 文档库的最新状态。如果要切换分支，可以在 Log Messages 对话框中，用鼠标右键单击 commit 节点，然后选择“Swtich/Checkout to (分支名称)”。

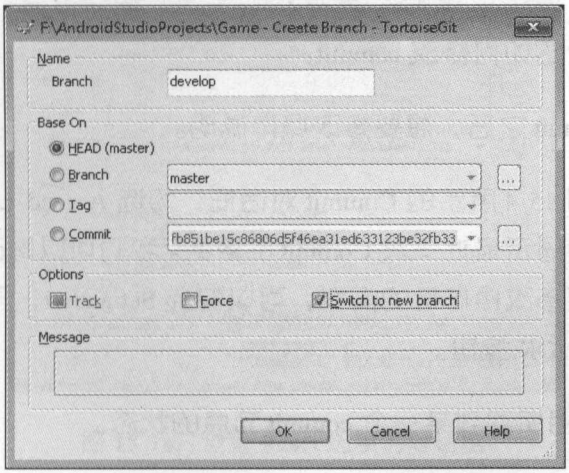


图 15-9 新增分支的对话框

补充说明

要如何知道当前是在哪一个分支呢？第一个方法是切换到 Windows 文件资源管理器，用鼠标右键单击程序项目的文件夹，调出如图 15-1 所示的菜单。如果文件夹中已经有了 Git 文档库，在图 15-1 矩形框中的第二个选项会变成“Git Commit -> (分支名称)”。其中的分支名称就是当前所在的分支。第二个方法是在如图 15-6 所示的 Log Messages 对话框左下方，取消勾选“All Branches”。对话框左上角就会显示当前所在的分支名称。

现在假设我们创建了一个名为 develop 的新分支，并且已经切换到这个新分支。接下来可以开始着手开发我们的程序项目。在每一个阶段中，使用前面介绍的方法，把程序项目的状态存入 Git 文档库。在修改程序项目期间，如果切换到如图 15-3 所示的 Working Tree 对话框，屏幕画面上会显示出有哪些文件被更改、加入或是删除了（如果原来对话框就已经打开，则必须单击 Refresh 按钮才会更新显示的内容）。在开发程序项目期间，不需要让 TortoiseGit 程序一直维持在运行的状态。我们可以随时将它关闭，等到下次要操作 Git 文档库的时候，再启动它。

接下来介绍如何修改 commit：

1. 执行 commit 之后，想要修改操作说明。

 打开如图 15-4 所示的 Commit 对话框，勾选 Amend Last Commit。对话框会显示最近一次 Commit 的数据。我们可以修改操作说明。如果想要修改操作者的资料，可以勾选 Set author，再进行编辑。最后单击 OK 按钮。

2. 让 Git 文档库回到某一个 commit 节点的状态。

 要完成这项工作有两种做法。就如同 Git 指令一样，可以使用 revert 或是 reset 的方式来实现。所谓 revert，就是不修改已经存入 Git

文档库的数据，而是把旧数据从指定的 commit 节点取出，这种做法类似于 checkout。要执行 revert 时，先打开 Log Messages 对话框。然后用鼠标右键单击要取出文件的 commit 节点，从快捷菜单中选择 Revert change by this commit，在下一个对话框中单击 OK 按钮。commit 节点中的文件就会被取出，并放到文件夹内。

第二种是 reset 的做法，也就是把 Git 文档库“消磁”，让其中某一部分的 commit 节点消失。要执行 reset 时，同样先打开 Log Messages 对话框。然后用鼠标右键单击要回到的目的 commit 节点，从快捷菜单中选择“Reset (分支名称) to this...”，屏幕画面会显示如图 15-10 所示的对话框。我们可以在对话框中挑选 Soft、Mixed 或是 Hard，这 3 种选项的效果和 Git 的 reset 指令一样。如果选择 Soft，表示只有文档库中的数据会更改，Git 索引和文件夹中的文件都不会受到影响。如果选择 Mixed，表示 Git 索引也会恢复到指定节点的状态，但是文件夹中的文件仍然不会受到影响。如果选择 Hard，则文档库、Git 索引和文件夹中的文件，都会恢复到指定节点的状态。

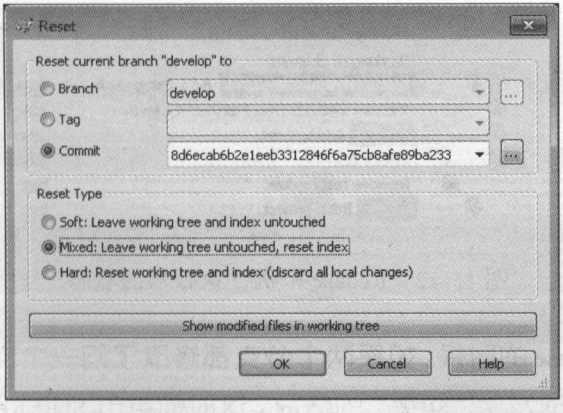


图 15-10　设置 Reset 选项的对话框

接下来介绍如何执行 Rebase。下面用一个例子来说明。请读者参见图 15-11。feature/fragment-based 是从 develop“长”出来的分支。当程

序项目在 feature/fragment-based 分支上修改时，develop 分支上的版本也进行了更新。在这种情况下，要把 develop 分支上的修改运用到 feature/fragment-based 分支。这样才能维持基础的一致性。也就是说，要让 feature/fragment-based 分支 Rebase 到 develop 分支上最新的 commit 节点。第一步是切换到 feature/fragment-based 分支，然后启动 Log Messages 对话框，用鼠标右键单击 develop 分支上最新的 commit 节点（表示要 Rebase 到这个节点），再从快捷菜单中选择“Rebase (分支名称) onto this...”。屏幕画面会显示一个对话框，单击对话框中的 Start Rebase 按钮。等到 Rebase 完成，再单击 Done 按钮即可。结果如图 15-12 所示。

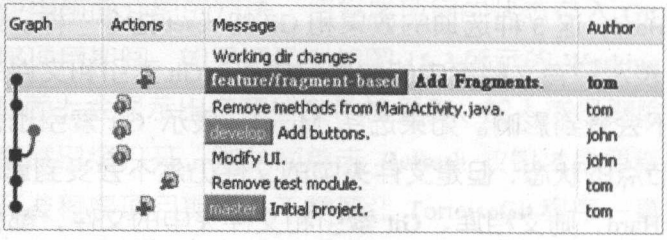


图 15-11　Rebase 范例：执行前的状态

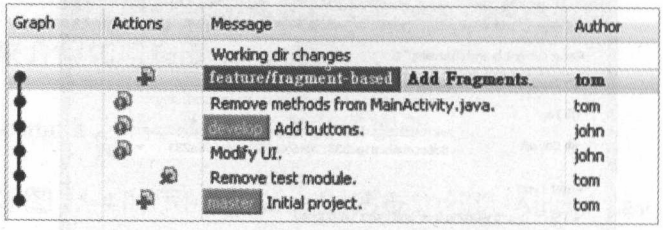


图 15-12　Rebase 范例：执行后的状态

执行 Rebase 的时候，如果两个分支都修改了同一个文件中相同的位置，就会出现发生“冲突”的情况。这时候单击 Start Rebase 按钮后，会显示出如图 15-13 所示的对话框。对话框下方会显示发生冲突的文件列表。用鼠标左键双击其中的文件，就会显示如图 15-14 所示的文件编辑窗口。上面的两个文件是要合并的原始文件，下面的文件是合

并后的结果。我们要对它进行编辑，完成之后单击上方工具栏的 Save 按钮。屏幕画面会出现一个对话框，单击其中的 Mark as resolved，表示已经处理好有冲突的内容，然后就可以关闭文件编辑窗口。按照同样的方式处理所有发生冲突的文件，最后单击 Commit 按钮，就可以完成 Rebase。如果发生冲突的时候，想要放弃 Rebase。可以在如图 15-13 所示的对话框中单击 Abort 按钮。文件夹中的文件和 Git 文档库都会恢复到还没执行 Rebase 时的状态。

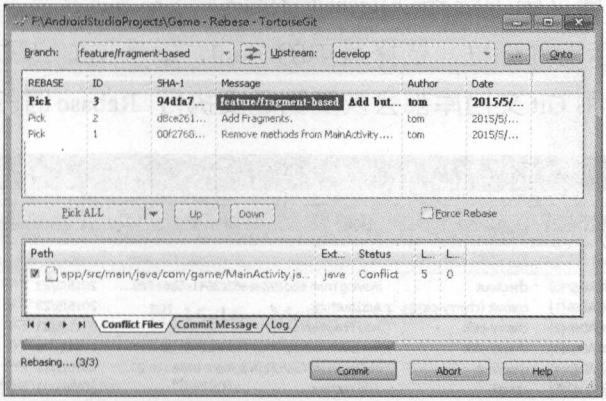


图 15-13　执行 Rebase 发生冲突的情况

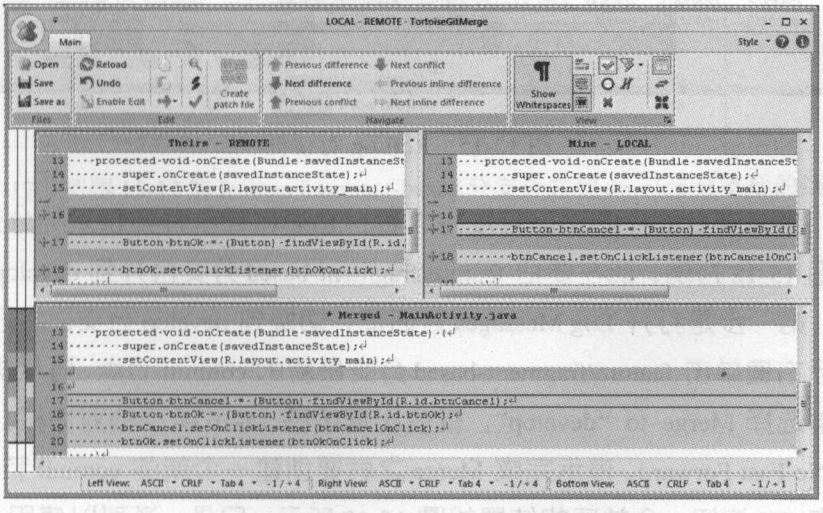


图 15-14　编辑发生冲突的文件

如果完成 Rebase 之后想要反悔，就恢复到还没执行 Rebase 时的状态。可以先调出如图 15-2 所示的快捷菜单，选择 Show Reflog，屏幕画面会出现如图 15-15 所示的对话框。Reflog 是 HEAD 变动的历史记录，读者可以参考第 9 单元的说明。可以从如图 15-15 所示的对话框中，找出执行 Rebase 之前 HEAD 的位置。执行 Rebase 通常会造成一连串 HEAD 的变动。所以也可以从最右边的 Date 字段，判断执行 Rebase 前 HEAD 所在的位置。确定 HEAD 的位置之后，用鼠标右键单击该位置，再从快捷菜单中选择“Reset (分支名称) to this...”，就会出现如图 15-10 所示的对话框。选择其中的 Hard，单击 OK 按钮即可。文件夹中的文件和 Git 文档库都会恢复到还没执行 Rebase 时的状态。

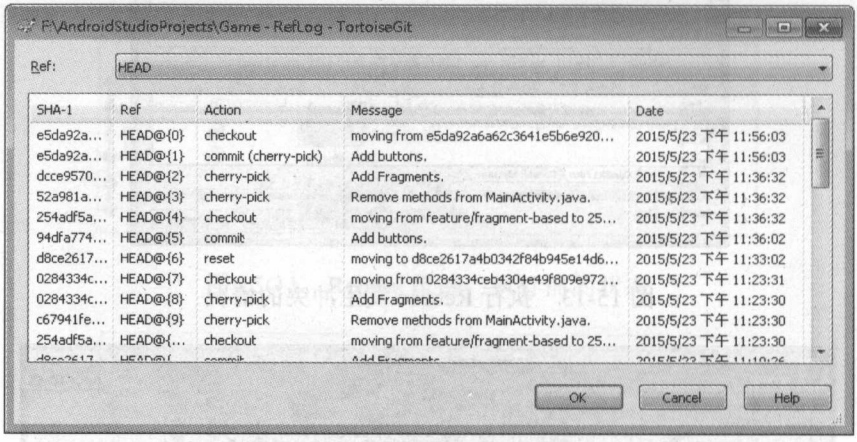


图 15-15　RefLog 对话框

当分支上的程序项目开发完成后，必须将它合并到原来的分支。以图 15-12 为例，现在要把 feature/fragment-based 分支合并到 develop 分支。第一步是打开 Log Messages 对话框，切换到 develop 分支。然后用鼠标右键单击 feature/fragment-based 分支最新的 commit 节点，再从快捷菜单选择 Merge to “develop”，就会出现如图 15-16 所示的对话框。勾选 No Fast Forward，表示完成 Merge 之后要创建一个新的 commit 节点，单击 OK 按钮。合并后的结果如图 15-17 所示。另外，还可以使用 Log

Messages 对话框显示 Git 文档库中的 commit 统计图。单击对话框下方的 Statistics 按钮就会出现如图 15-18 所示的窗口。右上角的下拉菜单可以选择数据类型。右下角的按钮可以切换图表种类。

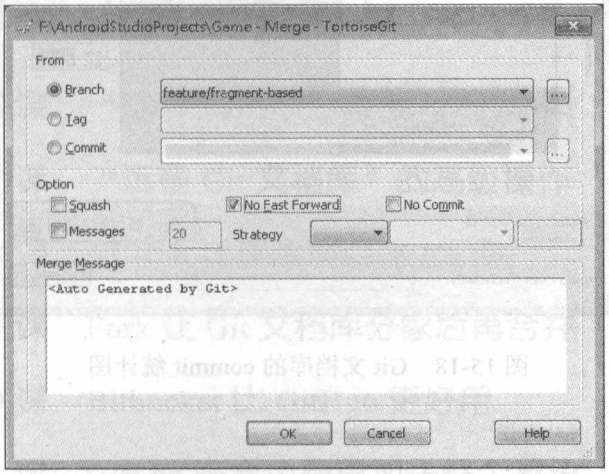


图 15-16　Merge 对话框

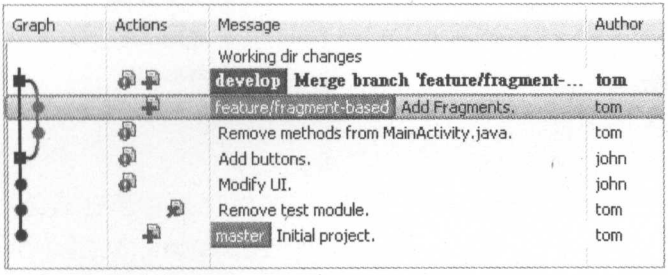


图 15-17　把图 15-12 的 feature/fragment-based 分支合并到 develop 分支后的结果

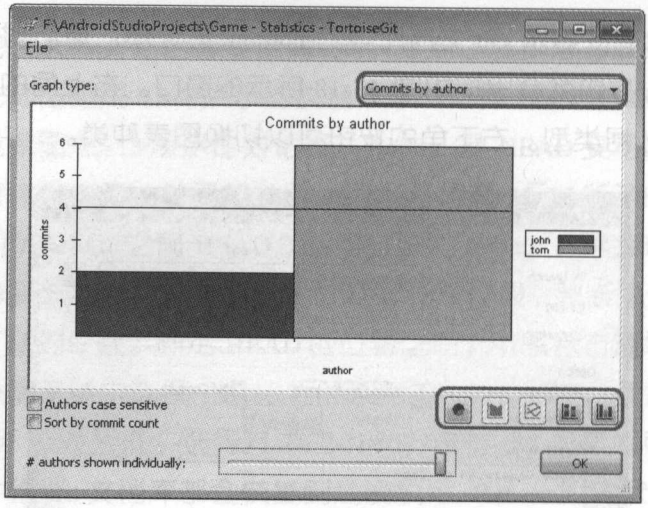


图 15-18　Git 文档库的 commit 统计图

第四部分　远程 Git 文档库和团队开关模式

第 16 课　“远程 Git 文档库”的基础操作

第 17 课　“远程 Git 文档库”的高级操作

第 18 课　GitHub 让程序项目飞上云端

第 19 课　Fork 让 Git 文档库分家后再合并

第 20 课　Bitbucket 比 GitHub 更好用

第 21 课　GitLab 完全免费再加送整个系统



第16课

“远程 Git 文档库”的基础操作

截至当前为止，我们操作的 Git 文档库都是存储在程序项目的文件夹内（也就是“.git”子文件夹内）。随着程序项目的进展，我们会把不同阶段的文件内容加入 Git 文档库。这个 Git 文档库是由我们直接操控，称它为“本地 Git 文档库（local repository）”。除了“本地 Git 文档库”以外，如果需要和其他人共同开发这个程序项目，还需要一个所谓的“远程 Git 文档库（remote repository）”。这个单元我们就来介绍“远程 Git 文档库”的功能和用法。

16-1 "远程 Git 文档库"的功能

程序项目和"本地 Git 文档库"，以及"远程 Git 文档库"之间的关系，可以用图 16-1 来说明。程序项目的内容是直接存入"本地 Git 文档库"，或是从"本地 Git 文档库"取出。而"远程 Git 文档库"的操作对象是"本地 Git 文档库"，而不是程序项目。如果程序项目是由一个人独立开发，而且不需要公开让别人取用。那么"本地 Git 文档库"就可以满足这样的需求。可是，如果程序项目需要提供给别人取用，或是采用团队开发模式，那么就需要创建一个共享的 Git 文档库。这个共享的 Git 文档库就是所谓的"远程 Git 文档库"，它和"本地 Git 文档库"的差异如图 16-1 所示。

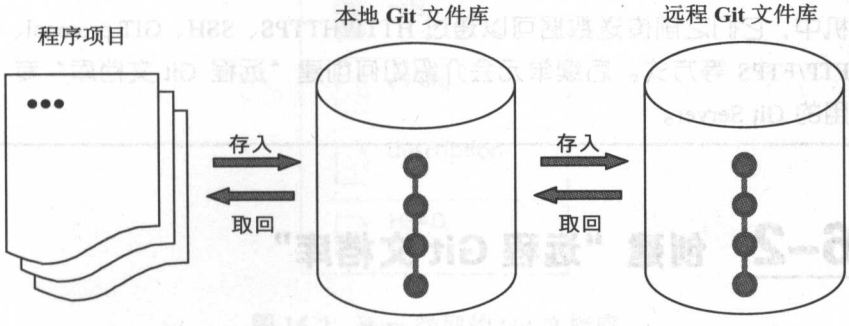


图 16-1 程序项目和"本地 Git 文档库"以及"远程 Git 文档库"之间的关系

1. "本地 Git 文档库"是存储在程序项目文件夹下的隐藏子文件夹".git"中（虽然可以借助配置文件来改变这项规则，但是通常不会这么做）。但是"远程 Git 文档库"却是直接存储在指定的文件夹内，而不是".git"子文件夹内。也就是说，"远程 Git 文档库"没有"现成的"程序项目（也就是 working tree）。这种模式的 Git 文档库被称为 Bare 类型。

2. 为了方便区分 Bare 类型的 Git 文档库，我们通常会在它的文件夹

名称最后加上扩展名“.git”，例如“game.git”。

3. 我们可以运用前面学过的 Git 指令，来查看 Bare 类型的 Git 文档库。但是通常不会直接修改它的内容，而是使用它对应的“本地 Git 文档库”来对它进行更新。

其实“远程 Git 文档库”不一定真的存储在另一台计算机中。我们也可以把它放在自己的计算机中。这个单元学习的重点是“本地 Git 文档库”和“远程 Git 文档库”之间如何互动。我们先不用在意二者之间如何传送数据。接下来我们就从“远程 Git 文档库”的创建方式开始介绍。

补充说明

如果“本地 Git 文档库”和“远程 Git 文件库”是存储在不同的计算机中，它们之前传送数据可以通过 HTTP/HTTPS、SSH、GIT protocol、FTP/FTPS 等方式。后续单元会介绍如何创建“远程 Git 文档库”专用的 Git Server。

16-2 创建“远程 Git 文档库”

在实际操作上，有可能先创建“本地 Git 文档库”，然后才决定要创建“远程 Git 文档库”。例如一开始的时候，先由某一位程序开发人员独立开发一个程序项目。经过一段时间之后，才决定投入更多人力共同开发。这时候，原来的程序开发人员，必须从他的“本地 Git 文档库”复制出一个公用的“远程 Git 文档库”。另外一种情况是在程序项目刚开始的时候，就创建好“远程 Git 文档库”。所有程序开发人员都从这个“远程 Git 文档库”，复制出自己的“本地 Git 文档库”，然后再着手进行开发。第二种方式比较简洁明了，因此我们先从这种方式开始介绍。

从上一个小节的讨论，我们已经知道，“远程 Git 文档库”是一个 Bare 类型的 Git 文档库。要创建 Bare 类型的 Git 文档库，只要在执行“git init”指令时，加上“--bare”选项即可，也就是：

```
git init --bare Git 文档库文件夹名称   ← 通常会加上扩展名“.git”
```

执行以上指令之后，如果我们查看 Git 文档库文件夹的内容，会看到如图 16-2 所示的结果。读者可以发现里面没有“.git”这个子文件夹。

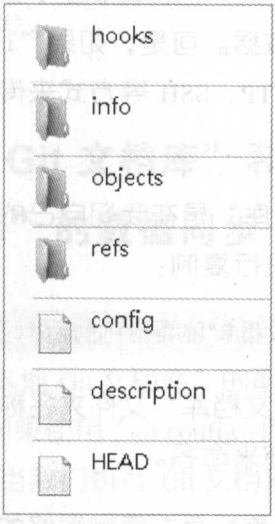


图 16-2　Bare 类型的 Git 文档库

补充说明

其实就技术上来说，一般类型的 Git 文档库也可以当成“远程 Git 文档库”。但是除非有特殊的原因，需要直接在“远程 Git 文档库”上开发程序项目。否则都会把“远程 Git 文档库”创建成 Bare 类型。

16-3 从“远程 Git 文档库”复制出“本地 Git 文档库”

现在要从“远程 Git 文档库”复制出一个“本地 Git 文档库”，并且存储在我们的计算机上。复制 Git 文档库是使用“git clone”指令，但是我们需要指定数据传送的方式。举例来说，如果“远程 Git 文档库”是在公司内部网络的某一台计算机上，那么可以使用“网络共享文件夹”的方式来传送数据。可是，如果“远程 Git 文档库”是在因特网上，就必须通过 HTTP、SSH 等方式来传送数据。我们分成下列几种情况说明：

1 步骤 如果“远程 Git 文档库”是在我们自己的计算机上，那么可以使用以下的指令格式进行复制：

```
git clone “远程 Git 文档库”的路径 “本地 Git 文档库”文件夹名称
```

如果省略“本地 Git 文档库”文件夹名称，Git 会用“远程 Git 文档库”的文件夹名称来命名。

2 步骤 如果“远程 Git 文档库”是在局域网络的计算机上，并且以“共享文件夹”的方式进行共享，那么可以采用以下指令格式：

```
git clone //计算机名称/“远程 Git 文档库”的路径 “本地 Git 文档库”文件夹名称
```

3 步骤 如果“远程 Git 文档库”是在 Web 服务器上，并且已经设置好 Web 服务器，让它能够执行 Git（后续单元会介绍如何架设这种类型的 Git Server）。我们可以使用以下指令复制 Git 文档库：

```
git clone http:// Web 服务器网址或是 IP 地址/“远程 Git 文档库”的路径 “本地 Git 文档库”文件夹名称
```

4 步骤 如果“远程 Git 文档库”是在 SSH 服务器上，并且已经创建一个 Git 程序专用的账号（后续单元会介绍如何安装和设置）。我们

可以使用以下指令复制 Git 文档库：

```
git clone Git 账号@SSH 服务器的网址或是 IP 地址:“远程 Git 文档库”的路径 “本地 Git 文档库”文件夹名称
```

复制成功之后，查看“本地 Git 文档库”文件夹的内容，会看到有一个名称叫作“.git”的子文件夹（提示：它是隐藏的文件夹，必须适当地设置文件夹查看模式，才能够看到它）。也就是说，这个复制得到的 Git 文档库是一般的 Git 文档库，而不是 Bare 类型，我们可以在上面进行程序项目的开发。

16-4 “本地 Git 文档库”和“远程 Git 文档库”的数据同步

“远程 Git 文档库”和复制得到的“本地 Git 文档库”之间存在着“父子”关系。在“本地 Git 文档库”的配置文件中会有一个 origin 属性，记录这个关系。如果使用“git config –l”指令来列出 Git 的设置，就会看到如下的内容。当我们执行 Git 文档库数据同步的操作时，会用到 origin 相关的属性。

```
… (其他属性)
remote.origin.url=“远程 Git 文档库”的地址
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
… (其他属性)
```

假设我们已经对“本地 Git 文档库”做了一些修改，现在要把这些修改，送到“远程 Git 文档库”。这项工作可以使用以下指令完成：

```
git push origin 分支名称
```

以上指令会把指定分支的最新状态，送到“远程 Git 文档库”存储。要特别说明的是，执行这个指令，不会在配置文件中记录“本地 Git

文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。但是如果加上“--set-upstream”选项（或是使用短选项“-u”）如下，Git 就会在配置文件中记录“本地 Git 文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。

```
git push --set-upstream origin 分支名称  ← “--set-upstream”选项可以换成短选项“-u”
```

执行上述指令之后，可以使用“git config –l”指令搭配“grep”指令，找出和该分支相关的设置：

```
git config -l | grep 分支名称
```

读者会看到如下的结果。这个对应关系会被“git push”指令使用，请读者参考“补充说明”。

```
branch.(分支名称).remote=origin                    ← 指定这个分支对应的“远程 Git 文档库”
branch.(分支名称).merge=refs/heads/develop  ← 指定这个分支所对应的分支名称
```

如果我们执行“git branch -a”指令，会看到和 origin 相关的分支如下：

```
* master                          ← 这个分支是在“本地 Git 文档库”
  remotes/origin/master  ← 这个分支是在“远程 Git 文档库”
```

remotes 开头的分支是用来追踪“远程 Git 文档库”的状态。由于“远程 Git 文档库”是多人共同使用，它的内容随时都有可能被其他人修改。如果发生这种情况，我们“本地 Git 文档库”中记录的“远程 Git 文档库”的状态，就会和最新的情况不一致。这是经常发生的情况，并不是错误。解决办法是，必须自己到“远程 Git 文档库”取得最新的状态。

补充说明

如果执行“git push”指令时，后面没有加上任何参数。屏幕画面会显示一段信息，提示我们需要在 Git 的配置文件中加入 push.default 的设置。我们可以执行下列指令，指定 push.default 为 matching：

```
git config --global push.default matching
```

这样的话，如果只执行“git push”指令，后面没有加上任何参数。Git 会把“本地 Git 文档库”中所有曾经传送给“远程 Git 文档库”的分支，都一并更新。或者，可以执行下列指令，指定 push.default 为 simple：

```
git config --global push.default simple
```

这样的话，如果只执行“git push”指令，后面没有加上任何参数。Git 会检查当前所在的分支是否在配置文件中记录了它对应的“远程 Git 文档库”的分支。如果有，就会传送更新给“远程 Git 文档库”。否则，就不会执行更新。和 matching 的设置相比之下，simple 是比较安全的做法。因为它只会更新当前所在的分支，不会一次更新全部的分支。

如果执行“git push --all”指令，“本地 Git 文档库”中所有的分支，都会传送到“远程 Git 文档库”。

如果我们执行“git push”指令的时候，显示以下错误信息，就表示“远程 Git 文档库”的状态已经被其他人修改。

```
! [rejected]        master -> master (fetch first)
error: failed to push some refs to '“远程 Git 文档库”'
hint: Updates were rejected because the remote contains work that you
do
hint: not have locally. This is usually caused by another repository
pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

这种情况就像是第 7 单元介绍的分支一样。下面用图 16-3 来说明。图 16-3 左边的情况是，在修改了“本地 Git 文档库”之后，很幸运地，没有其他人修改“远程 Git 文档库”（其实这就像是第 8 单元图 8-2 的情况）。这时候如果执行“git push”指令，Git 会把“本地 Git 文档库”的内容，传送到“远程 Git 文档库”，并进行合并。这种情况的合并不会发生冲突，因此不会有任何问题。

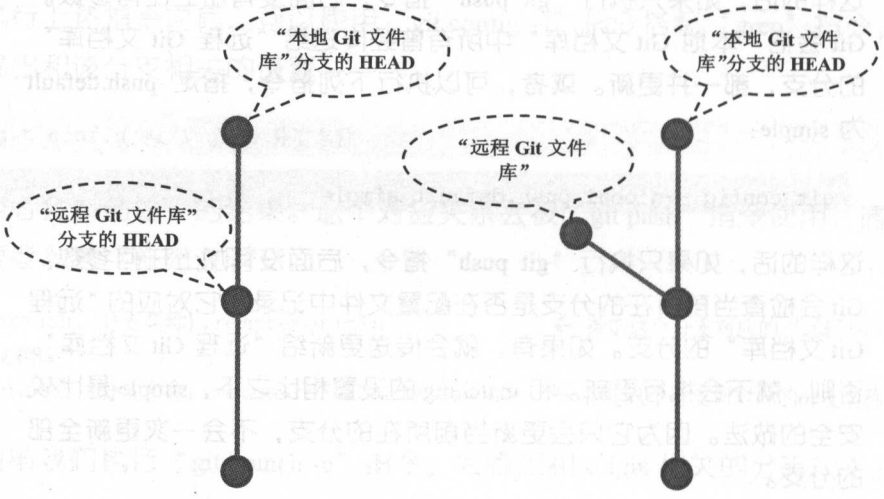


图 16-3　“本地 Git 文档库”和“远程 Git 文档库”可能出现的两种情况

图 16-3 右边的情况是，在修改“本地 Git 文档库”的时候，有其他人更新了“远程 Git 文档库”。这时候如果执行“git push”指令，Git 则会发现无法使用第 8 单元介绍的 fast-forward merge 来更新“远程 Git 文档库”，而必须使用 3-way merge，这种合并方式就有可能发生冲突，因此向我们报告前面的错误信息。处理这种情况的方法是先执行以下指令：

```
git pull
```

"git pull" 指令会执行两项工作：

1. 从 "远程 Git 文档库" 取回当前所在分支的最新数据。完成这项工作之后，计算机上的 "远程 Git 文档库" 的分支状态就和实际的情况一致。如果想要一次取得全部分支最新的数据，可以加上 "--all" 选项。
2. 把 "远程 Git 文档库" 的分支合并到 "本地 Git 文档库" 的分支。

执行第 2 个步骤有可能出现冲突的情况，这时候会显示以下信息：

```
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From (“远程 Git 文档库”)
   0e2bad3..85dc203  master     -> origin/master
Auto-merging (文件名)
CONFLICT (content): Merge conflict in (文件名)
Automatic merge failed; fix conflicts and then commit the result.  ← 发生冲突
```

处理冲突的方法和第 8 单元介绍的完全一样。合并成功之后，会变成如图 16-4 所示的结果（提示：如果合并之后想要反悔，可以参考第 8 单元的做法，把 Git 文档库和文件夹中的文件恢复到合并前的状态）。执行 "git pull" 指令之后，可以再次执行 "git push" 指令。现在，"本地 Git 文档库" 的内容就可以成功地上传到 "远程 Git 文档库"，而且 "远程 Git 文档库" 也会看到执行合并的记录。

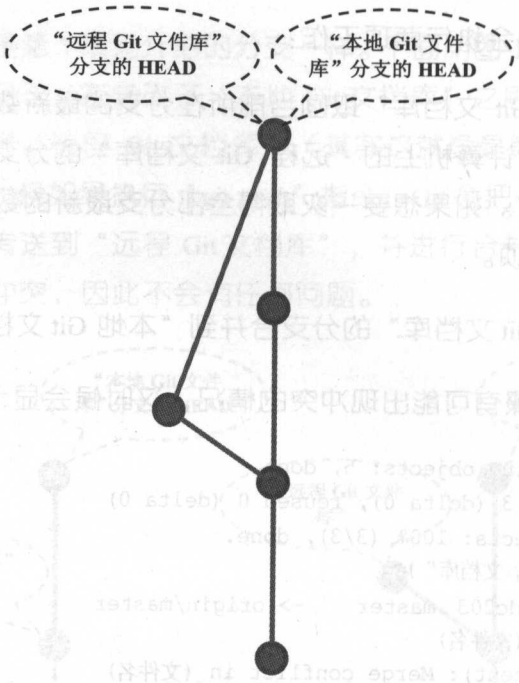


图 16-4　将图 16-3 右边的状态执行合并后的结果

最后再讨论一下“git pull”指令执行的工作。它的第一个步骤是更新我们计算机上“远程 Git 文档库”的分支的状态，让它和“远程 Git 文档库”的最新状态一致。其实这项工作可以借助另一个指令来完成，那就是“git fetch”。也就是说，“git pull”指令其实就是先执行“git fetch”，接着再执行“git merge”（提示：执行“git fetch”时，如果加上“--all”选项，可以一次更新全部分支）。现在要考验一下读者的记忆，请回想一下“git merge”和“git rebase”的差别（提示：在第 9 单元可以找到 Rebase 的说明）。从 commit 节点演进图来看，合并会造成两个分支交错的网状架构，但是 Rebase 不会。Rebase 只会让 commit 节点演进图以直线的方式延展，因此在查看的时候会简单许多。如果执行“git pull”指令，它会使用“git merge”来合并“远程 Git 文档库”和“本地 Git 文档库”的分支。但是如果加上“--rebase”选项（或是使用短选项“-r”）如下：

```
git pull --rebase   ← 也可以把“--rebase”换成“-r”
```

就会用“git rebase”指令取代原来的“git merge”指令。或者我们也可以按序执行下列指令，达到完全相同的效果。图 16-5 是使用 Rebase 得到的结果，读者可以比较它和图 16-4 的差别。

```
git fetch                    ← 等于“git pull”指令的第一项工作
git rebase origin/master ← 用“git rebase”指令取代“git merge”
```

如果执行“git rebase”指令的时候发生了冲突，可以使用第 9 单元介绍的方法来处理。如果在 Rebase 之后想要反悔，也可以参考第 9 单元的说明，把 Git 文档库和文件夹中的文件恢复到 Rebase 之前的状态。

补充说明

1. 我们不一定要把“本地 Git 文档库”的每一个分支都送到“远程 Git 文档库”存储。“远程 Git 文档库”的分支是让多人共同开发使用。如果只是自己私底下使用的分支，就不需要传送到“远程 Git 文档库”。
2. 前面曾经解释过，一般类型的 Git 文档库也可以当成“远程 Git 文档库”，不一定非得要用 Bare 类型的文档库不可。但是，读者可以思考一下，如果把一般 Git 文档库当成“远程 Git 文档库”，会发生什么情况。由于“远程 Git 文档库”会被其他人执行的“git push”指令所修改，而且在一般 Git 文档库的文件夹内包含了被取出的程序项目（也就是 working tree）。如果文档库的内容被其他人修改，working tree 中的程序项目版本容易和文档库的内容出现混淆的情况。

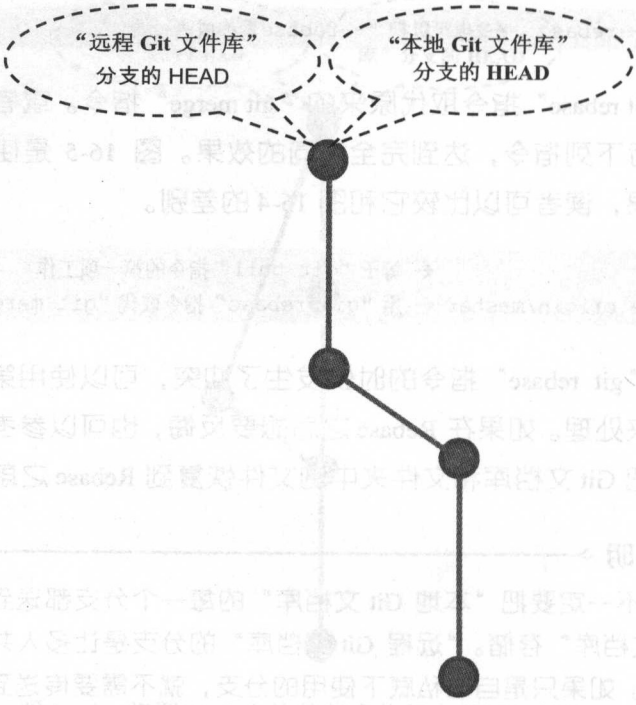


图 16-5　将图 16-3 右边的状态执行 Rebase 后的结果



第17课

“远程 Git 文档库”的高级操作

前一个单元的情况是先创建一个 Bare 类型的“远程 Git 文档库”，再从它复制出“本地 Git 文档库”。这是团队开发模式的标准做法。只不过有时候程序项目并不是一开始就由团队共同开发，而是先由某一位程序开发人员起头。也就是说，程序项目的 Git 文档库是在某一个人的计算机中。当其他人要加入时，必须从这个私有的 Git 文档库复制出共享的“远程 Git 文档库”。再按照上一个单元的做法，开始进行团队开发。

要从“本地 Git 文档库”复制出 Bare 类型的“远程 Git 文档库”，可以使用以下指令：

```
git clone --bare 程序项目文件夹名称 “远程 Git 文档库”的路径
```

但是只做这些是不够的！因为我们是反过来从“本地 Git 文档库”复制出“远程 Git 文档库”。这种方式不会在“本地 Git 文档库”的配置文件中留下 origin 属性。换句话说，“本地 Git 文档库”不知道它和“远程 Git 文档库”之间的关系，因此接下来我们要自己设置这个关系。要完成这项工作需要使用“git remote”指令，格式如下：

```
git remote add “远程 Git 文档库”的名称 “远程 Git 文档库”的 url
```

只要我们把“远程 Git 文档库”命名为 origin，就会得到和上一个单元相同的结果。但是到这里为止，我们只是设置了文档库之间的对应关系，还没有创建“远程 Git 文档库”的追踪分支。如果执行“git branch -a”指令，只会看到“本地 Git 文档库”的分支，没有任何“远程 Git 文档库”的追踪分支。因此还要执行下列指令，让 Git 自动对比“本地 Git 文档库”和“远程 Git 文档库”的分支名称，找出对应的分支，并且在“本地 Git 文档库”中创建追踪分支。

```
git remote update
```

补充说明

1. 其实也可以用一个变通的办法，让 Git 自动创建“本地 Git 文档库”和“远程 Git 文档库”之间的对应关系。就是在复制出“远程 Git 文档库”之后，删除计算机上的程序项目。再按照上一个单元的做法，从“远程 Git 文档库”复制出“本地 Git 文档库”。
2. 即使“本地 Git 文档库”中没有 origin 属性，也可以执行“git push”指令，只要使用以下格式即可：

```
git push “远程 Git 文档库”的 url 分支名称
```

“git remote add”这个指令让我们发现两件事情：

1. “远程 Git 文档库”的名称不一定要叫作 origin，使用其他名称。
2. “远程 Git 文档库”也不是唯一的，我们可以在“本地 Git 文档库”中加入其他对应的“远程 Git 文档库”。

以图 17-1 来说，假设 origin 这个“远程 Git 文档库”原来就已经存在。现在我们要创建另一个“远程 Git 文档库”，并且把 master 和 bug 这两个分支送到这个新的“远程 Git 文档库”中。要完成这项工作，只需要按序执行以下步骤。首先在远程计算机上创建一个 Bare 类型的 Git 文档库：

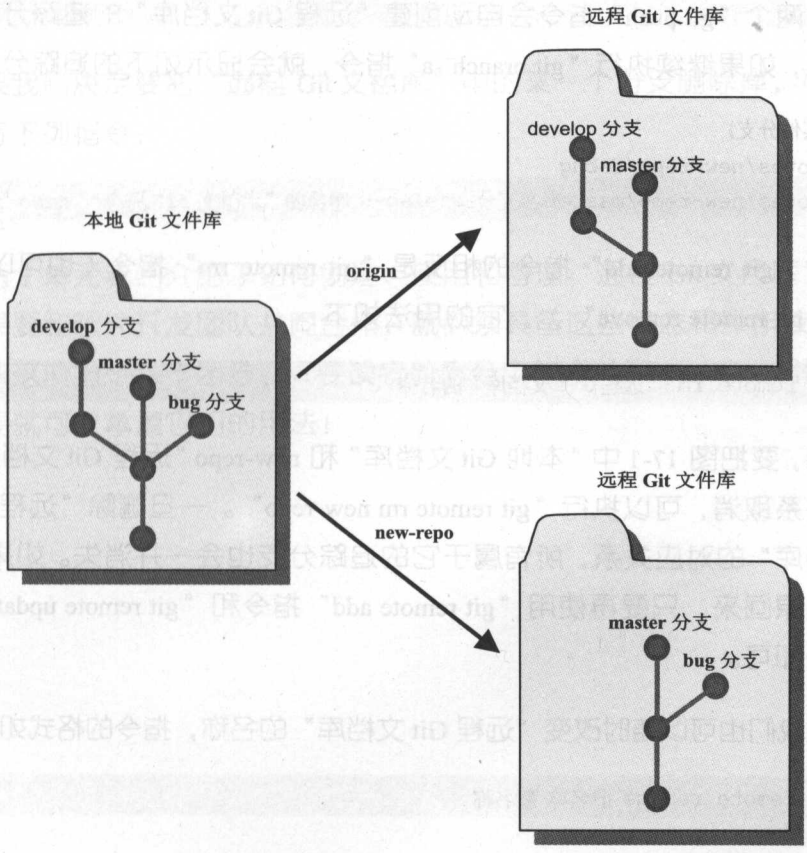


图 17-1 “本地 Git 文档库”对应到多个“远程 Git 文档库”的例子

```
git init --bare Git 文档库的文件夹名称
```

接下来在自己的计算机上执行下列指令：

```
git remote add new-repo “远程 Git 文档库”的 url　← new-repo 是我们自己取的名称
git push -u new-repo master
git push -u new-repo bug
```

以上两个“git push”指令会自动创建“远程 Git 文档库”的追踪分支。因此，如果继续执行“git branch -a”指令，就会显示如下的追踪分支：

```
…(其他分支)
remotes/new-repo/debug
remotes/new-repo/master
```

“git remote add”指令的相反是“git remote rm”指令（也可以写成“git remote remove”），它的用法如下：

```
git remote rm “远程 Git 文档库”的名称
```

例如，要把图 17-1 中“本地 Git 文档库”和 new-repo“远程 Git 文档库”的关系取消，可以执行“git remote rm new-repo”。一旦删除“远程 Git 文档库”的对应关系，所有属于它的追踪分支也会一并消失。如果要再还原回来，只要再使用“git remote add”指令和“git remote update”指令即可。

我们也可以随时改变“远程 Git 文档库”的名称，指令的格式如下：

```
git remote rename 旧名称 新名称
```

改变“远程 Git 文档库”的名称之后，和它相关的远程追踪分支的名称也会自动更新。另外也可以改变“远程 Git 文档库”的 url：

```
git remote set-url “远程 Git 文档库”的名称 新的 url
```

如果要显示某一个“远程 Git 文档库”的详细信息，可以执行下列指令：

```
git remote show “远程 Git 文档库”的名称
```

如果要列出“本地 Git 文档库”对应的所有“远程 Git 文档库”，可以执行下列指令：

```
git ls-remote
```

也可以使用以下指令显示和“远程 Git 文档库”相关的设置：

```
git remote -v
```

如果我们决定要把“远程 Git 文档库”中的某一个分支删除掉，可以执行下列指令：

```
git push “远程 Git 文档库”的名称 --delete 分支名称
```

这两个单元我们介绍了如何创建、使用和管理“远程 Git 文档库”。如果要和软件开发团队共同合作，就必须具备这些概念和操作技巧。其实这些指令并不困难，只要读者熟悉分支的合并和 Rebase，相信很容易就可以掌握它们的用法！



第18课

GitHub 让程序项目飞上云端

在因特网上，有一些网站专门提供存储 Git 文档库的服务。当前最知名的包括 GitHub、Bitbucket 和 GitLab，其中又以 GitHub 拥有最多的用户，但是它提供的“优惠”却比较少。没有付费的用户，只能够创建公开的 Git 文档库。也就是说，全世界的人都可以自由下载他的程序项目。Bitbucket 和 GitLab 的免费用户可以创建私有的 Git 文档库，因此对于想要保护知识产权的用户来说，有比较好的保障。这些网站的功能和操作流程都很类似。这个单元我们先从 GitHub 开始介绍。

18-1 注册和设置 GitHub

要使用 GitHub 提供的服务，必须先到 GitHub 网站完成注册。开启网页浏览器，用百度或者 Google 搜索 GitHub，就可以找到它的官方网站。进入 GitHub 官方网站会看到如图 18-1 所示的画面。如果已经注册过，可以单击屏幕画面右上方的 Sign in 按钮，切换到登录画面。如果是第一次使用，直接在屏幕画面右边的三个空白字段输入用户名称、email 账号和密码，再单击 Sign up for GitHub，就会切换到如图 18-2 所示的屏幕显示画面。中间有一个表格让我们挑选使用模式。最下面是 Free 模式，我们不需要付费，但是所创建的 Git 文档库是公开的，全世界的人都可以下载。挑选好使用模式之后，单击最下面的 Finish sign up，就会进入如图 18-3 所示的用户操作首页。

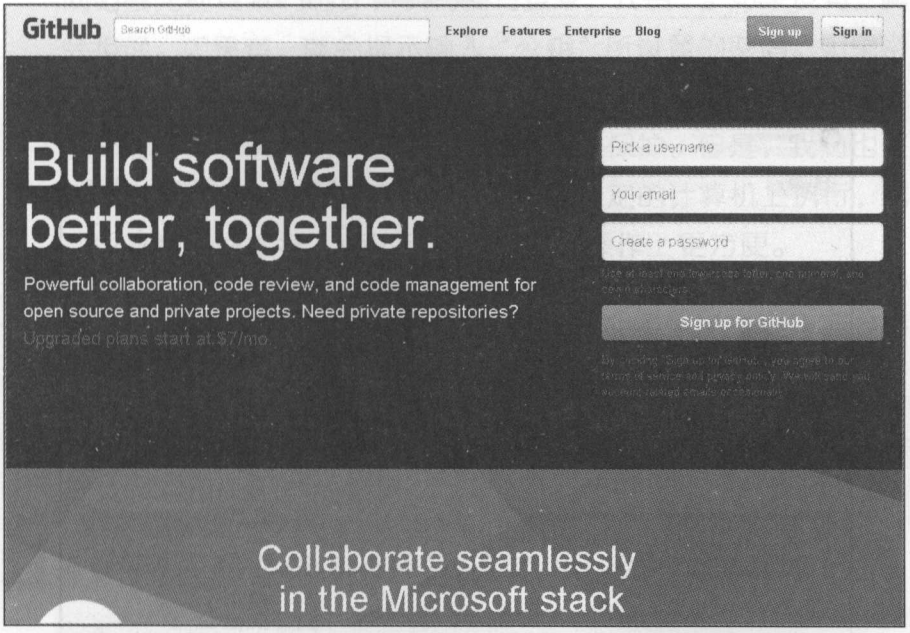


图 18-1 GitHub 网站首页



图 18-2　选择使用模式

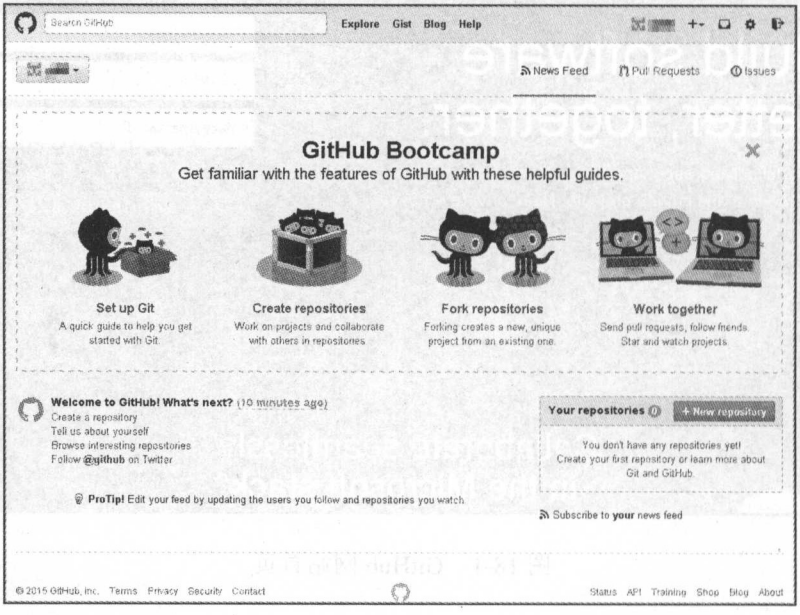


图 18-3　GitHub 的用户操作首页

GitHub网站上的Git文档库是扮演“远程Git文档库”的角色。计算机上的“本地Git文档库”可以通过两种方式存取GitHub网站上的“远程Git文档库”。第一种是使用HTTPS通信协议，另一种则是使用SSH通信协议。它们的优点如下：

1. HTTPS和SSH通信协议在传送数据的过程中都会进行加密，因此是很安全的数据传输方式。

2. 如果使用HTTPS把数据上传到GitHub网站上的“远程Git文档库”，Git程序会要求输入GitHub网站的账号和密码。

3. 如果使用SSH，我们必须按照后续的说明，在计算机上创建一对“密钥”，然后把其中的“公钥”，加入GitHub网站的账号中。当“本地Git文档库”的数据，要通过SSH上传到GitHub时，GitHub网站会先检查我们的计算机有没有登录“公钥”。如果没有，就会拒绝。如果有，就会提示输入“公钥”所设置的密码。如果密码正确，才会执行上传数据的操作。也就是说，SSH通信协议提供“限定计算机”和“验证密码”这两道保护。但是，我们也可以不对“密钥”设置密码，这样只要在特定的计算机上执行，就可以直接传送数据，而不用输入密码，操作上更方便。

4. 我们可以随时切换使用HTTPS或是SSH通信协议，操作上有很大的弹性。

接下来我们介绍如何在计算机上创建“密钥”，并且把对应的“公钥”登录到GitHub网站的账号：

1 步骤 在计算机上执行Git Bash程序，输入以下指令：

```
ssh-keygen
```

程序会询问存储“密钥”文件的路径，直接按 Enter 键，接受预订的路径即可，然后输入两次密码或是直接按 Enter 键。这个密码就是后续要将“本地 Git 文档库”的数据传送给 GitHub 网站时，需要输入的密码。如果直接按 Enter 键，后续操作就不需要验证密码。

```
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/(用户账号)/.ssh/id_rsa):
←直接按 Enter
Enter passphrase (empty for no passphrase): ← 输入密码，或是直接按
Enter 键
Enter same passphrase again: ← 再次输入密码，或是直接按 Enter 键
```

2 步骤 使用文件资源管理器查看“c:\用户\(用户账号)\.ssh”文件夹，会在里面找到两个文件：id_rsa 文件是“私钥”；id_rsa.pub 文件是“公钥”。用文字编辑程序打开“公钥”文件，复制全部内容。

3 步骤 开启网页浏览器，登录 GitHub 网站。然后在如图 18-3 所示的右上角，单击 Settings 按钮（小齿轮图标）。

4 步骤 浏览器会显示如图 18-4 所示的屏幕画面，先单击左边选项的 SSH keys，再选择右上角的 Add SSH key。在 Title 字段输入自定义的名称，然后在 Key 字段贴上步骤 2 复制的“公钥”文件内容。最后单击 Add key 按钮。屏幕画面上会新增一个 key 选项。

我们可以按照同样的方式，加入其他计算机的 SSH“公钥”。提醒读者，同一台计算机上，不同用户账号的 SSH“密钥”是独立的。也就是说，账号 A 创建的“密钥”不能让账号 B 使用。

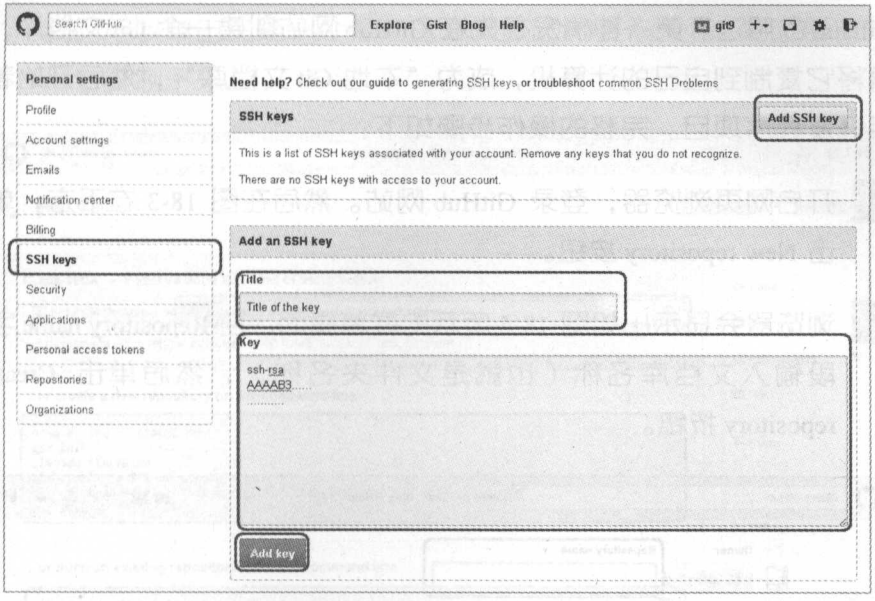


图 18-4　登录 SSH“公钥”的操作界面

18–2　使用 GitHub 的“远程 Git 文档库”

其实 GitHub 的“远程 Git 文档库”的用法，和第 16 单元以及第 17 单元介绍的操作方式几乎一模一样。我们学过的指令都可以使用，操作逻辑也完全相同。以下是 GitHub 网站提供的功能：

1. 我们可以随意创建新的 Git 文档库，以及删除已经存在的 Git 文档库。
2. GitHub 网站提供了一个独立执行的程序，可以进行简单的 Git 操作以及文档库同步的功能。
3. GitHub 网站提供图表统计功能，帮助我们了解 Git 文档库更新的情况。

接下来我们按照第 16 单元和第 17 单元讨论的两种使用情况来介绍

GitHub 的用法。第一种情况是先在 GitHub 网站创建一个 Git 文档库，再将它复制到自己的计算机，成为“本地 Git 文档库”，然后开始着手开发程序项目。完整的操作步骤如下：

1 步骤 开启网页浏览器，登录 GitHub 网站。然后在图 18-3 右下方，单击 New repository 按钮。

2 步骤 浏览器会显示出如图 18-5 所示的屏幕画面。在 Repository name 字段输入文档库名称（也就是文件夹名称），然后单击 Create repository 按钮。

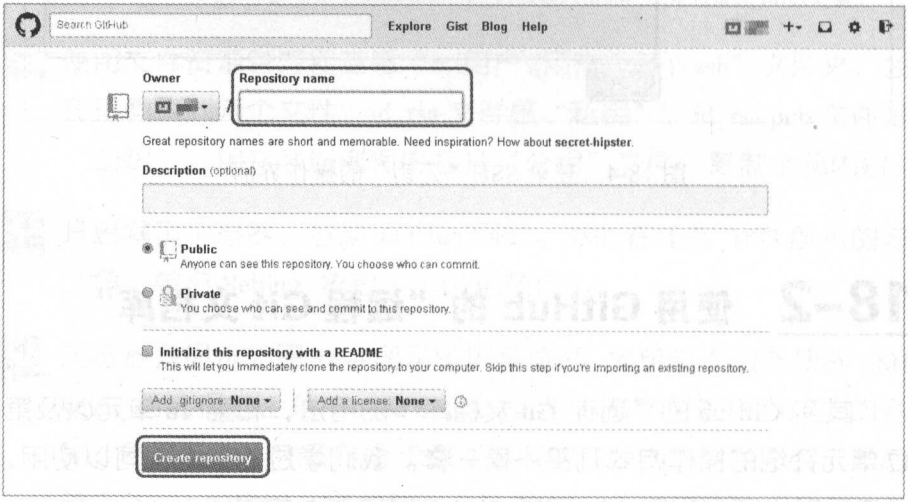


图 18-5　创建 Git 文档库

3 步骤 浏览器会显示出如图 18-6 所示的屏幕画面。这个屏幕画面提示我们如何使用 Git 指令，将计算机上的“本地 Git 文档库”上传到 GitHub。我们可以单击最上面、在第一个框选择区域中的 HTTPS 或是 SSH 按钮。屏幕画面中的 Git 指令会随着变动。这里的指令范例是告诉我们，如何使用 HTTPS 或是 SSH 来传送数据。这些指令我们已经在第 16 单元和第 17 单元介绍过，读者应该能够了解它们的功能（提示：可以参考本书附录的 Git 指令和使用说明）。

我们可以单击在第一个框选区域最右边的复制按钮，把文档库的网址复制下来。

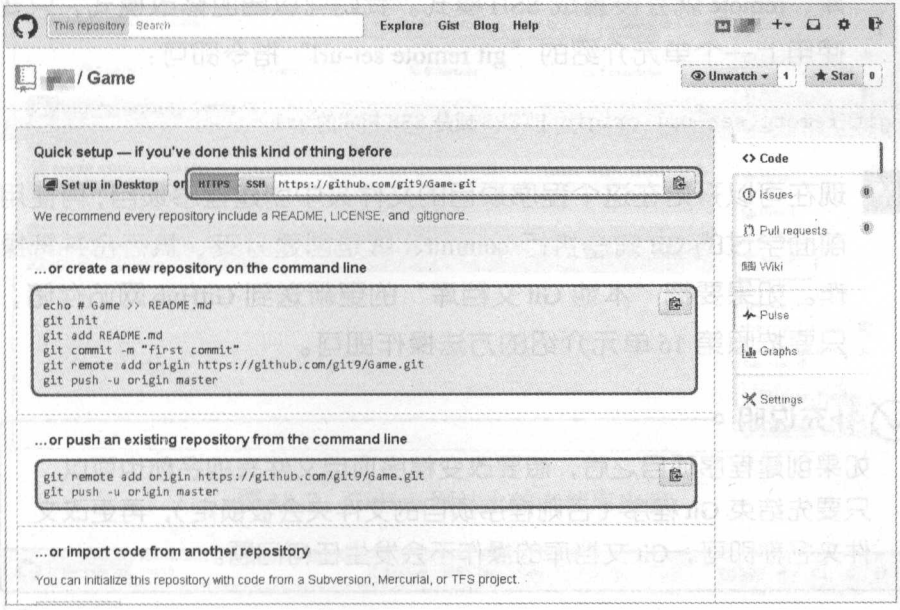


图 18-6　创建 Git 文档库之后的操作提示画面

4 步骤 单击网页画面左上角的小图标回到操作首页。网页右下方会显示刚刚创建的 Git 文档库。

5 步骤 接下来我们要从 GitHub 网站复制 Git 文档库到自己的计算机中，成为“本地 Git 文档库”。启动 Git Bash 程序，切换到程序项目的文件夹，输入以下指令（提示：要在 Git Bash 程序中粘贴文字时，先单击程序屏幕画面左上角的图标，再选择“编辑 > 粘贴”）。从 GitHub 网站复制 Git 文档库时不需要输入密码，因为我们的程序项目是完全公开的，所有人都可以下载。

```
git clone（粘贴步骤 3 复制的文档库的网址）程序项目文件夹名称
```

6 步骤 复制完成后，切换 Git Bash 程序的工作目录，进入程序项目的文件夹，然后执行“git remote –v”指令，就会看到 remote 相关的设

置。如果前一个步骤是以 HTTPS 的方式复制文档库，remote 就会设置成 HTTPS 模式。如果前一个步骤是以 SSH 的方式复制文档库，remote 就会设置成 SSH 模式。我们可以随时修改模式，只要使用上一个单元介绍的“git remote set-url”指令即可：

```
git remote set-url origin HTTPS 或是 SSH 模式的 url
```

7 步骤 现在可以开始在这个程序项目的文件夹中创建程序项目，再使用前面学过的 Git 指令执行 commit、或是创建分支、执行合并等操作。如果要把“本地 Git 文档库”的更新送到 GitHub 网站存储，只要按照第 16 单元介绍的方法操作即可。

补充说明

如果创建程序项目之后，想要改变程序项目文件夹的名称也可以，只要先结束 Git 程序（否则程序项目的文件夹会被锁定），再更改文件夹名称即可，Git 文档库的操作不会发生任何问题。

8 步骤 如果要删除 GitHub 网站上的文档库，可以从如图 18-3 所示的操作首页右下方单击文档库名称，就会显示出如图 18-7 所示的画面。单击右边的 Settings 选项，然后在下一个操作界面最下方，单击 Delete this repository。网页会显示一个对话框，要求输入文档库名称。输入完毕后，单击下方的按钮，就可以删除文档库。

9 步骤 如果想要查看 Git 文档库的统计资料，可以从图 18-7 右边的选项中选择 Graphs，就会出现如图 18-8 所示的图表。屏幕画面上方可以切换不同的统计数据。

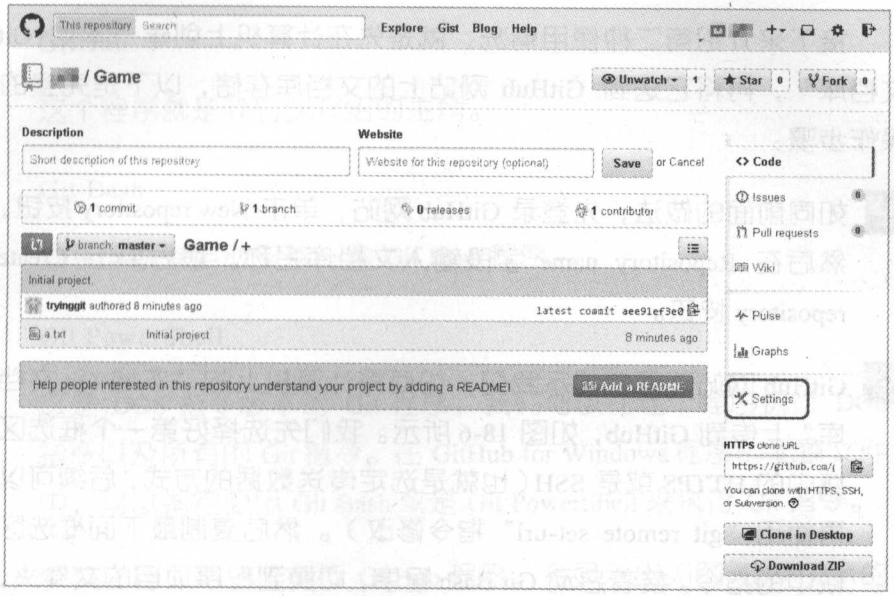


图 18-7　GitHub 网站上的文档库内容

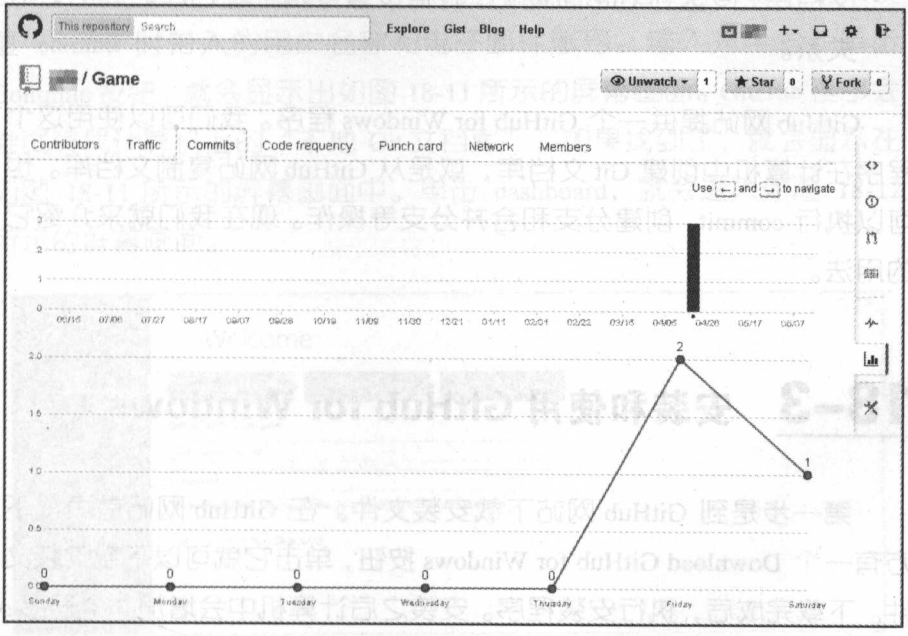


图 18-8　GitHub 网站的文档库使用统计图

接下来介绍第二种使用情况，就是先在计算机上创建“本地 Git 文档库”，再将它送到 GitHub 网站上的文档库存储，以下是完整的操作步骤：

1 步骤 如同前面的做法，先登录 GitHub 网站，单击 New repository 按钮，然后在 Repository name 字段输入文档库名称，最后单击 Create repository 按钮。

2 步骤 GitHub 网站画面会提示我们，如何将计算机上的“本地 Git 文档库”上传到 GitHub，如图 18-6 所示。我们先选择好第一个框选区域中的 HTTPS 或是 SSH（也就是选定传送数据的方式，后续可以再使用“git remote set-url”指令修改）。然后复制最下面框选区域中的指令。接着启动 Git Bash 程序，切换到程序项目的文件夹，执行刚刚复制的指令。就可以把程序项目文件夹中的“本地 Git 文档库”传送到 GitHub 网站，并且设置好两个文档库之间的对应关系。

GitHub 网站提供一个 GitHub for Windows 程序。我们可以使用这个程序在计算机中创建 Git 文档库，或是从 GitHub 网站复制文档库。也可以执行 commit、创建分支和合并分支等操作。现在我们就来介绍它的用法。

18-3 安装和使用 GitHub for Windows

第一步是到 GitHub 网站下载安装文件。在 GitHub 网站首页最下方有一个 Download GitHub for Windows 按钮，单击它就可以下载安装文件。下载完成后，执行安装程序。安装之后计算机中会增加 3 个程序：

1. GitHub

这个程序就是我们要介绍的主角。

2. Git Bash

安装程序会自动安装一套 Git Bash 程序。

3. Git PowerShell

这是 DOS 指令版本的 Git 程序，我们可以在这个程序执行 DOS 指令以及所有的 Git 指令。在 GitHub for Windows 程序的配置文件中，可以指定使用 Git Bash 或是 Git PowerShell 来执行 Git 指令。

安装之后，第一次执行 GitHub 程序，会显示出如图 18-9 所示的屏幕画面。我们必须填入 GitHub 网站的账号和密码，然后单击 Log in 按钮，就会切换到如图 18-10 所示的屏幕画面。这个屏幕画面用来设置 commit 时带入的用户名称和电子邮件账号。输入完毕后，单击 Continue 按钮，就会显示出如图 18-11 所示的屏幕画面。GitHub 程序会尝试寻找计算机上的“本地 Git 文档库”，如果找到了，就会显示在如图 18-11 所示的屏幕画面中。单击 dashboard，就会进入如图 18-12 所示的屏幕画面。

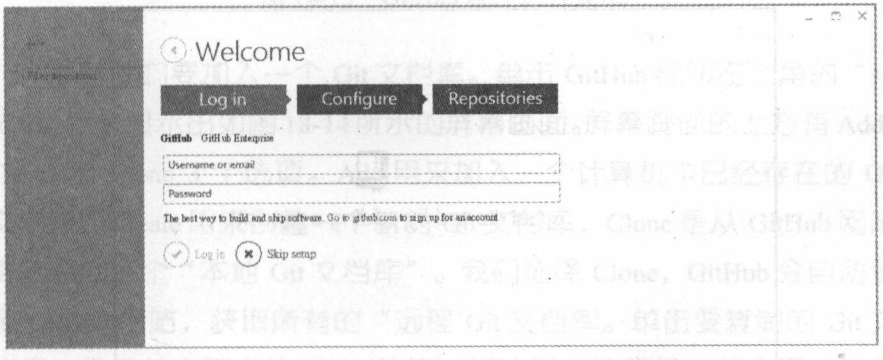


图 18-9 第一次启动 GitHub 程序的屏幕画面

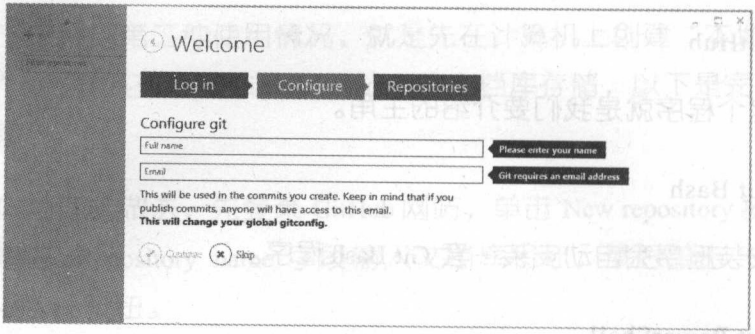


图 18-10　设置 commit 时带入的用户名称和电子邮件账号



图 18-11　GitHub 程序尝试寻找计算机上的“本地 Git 文档库”

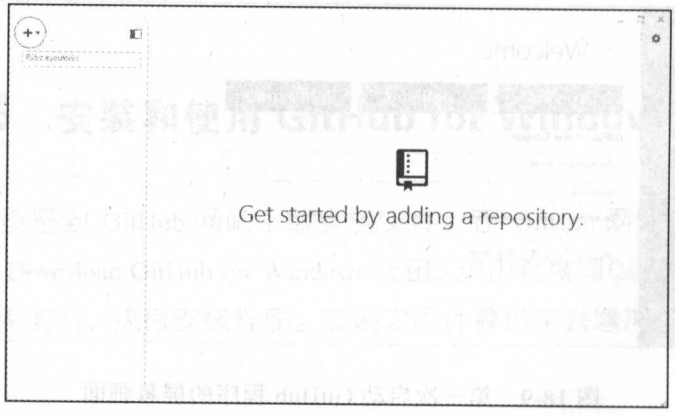


图 18-12　没有加入任何 Git 文档库的 GitHub 程序之屏幕显示画面

在开始操作以前，先来介绍一下 GitHub 程序的设置。单击图 18-12 右上角的“小齿轮”图标，然后选择 Options，就会出现如图 18-13 所示的画面。左上方的 Accounts 字段就是前面输入的 GitHub 网站账号。左下方的 Configure git 字段就是前面输入的 commit 用户名称和电子邮件账号。右上方的 Clone path 是从 GitHub 网站复制文档库时默认的存储路径。右下方的 Default shell 是指定使用 Git Bash 或是 Git PowerShell 来执行 Git 指令。设置完毕后，单击左下角的 Save 按钮即可。

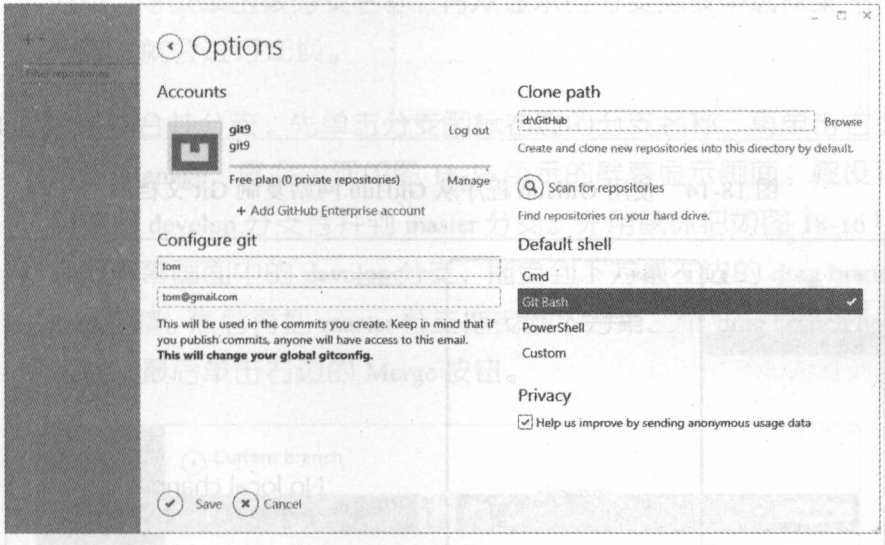


图 18-13　GitHub 程序的设置项

现在我们要加入一个 Git 文档库。单击 GitHub 程序左上角的“+”按钮，就会显示出如图 18-14 所示的屏幕画面。屏幕画面的上方有 Add、Create 和 Clone 3 个选项。Add 用来加入一个计算机中已经存在的 Git 文档库，Create 用来创建一个新的 Git 文档库，Clone 是从 GitHub 网站复制得到一个“本地 Git 文档库”。我们选择 Clone，GitHub 会自动登录 GitHub 网站，获取所有的“远程 Git 文档库。单击要复制的 Git 文档库，然后单击下方的 Clone 按钮，再选择存储路径，就会开始复制文档库。复制完成后，会显示如图 18-15 所示的屏幕画面。

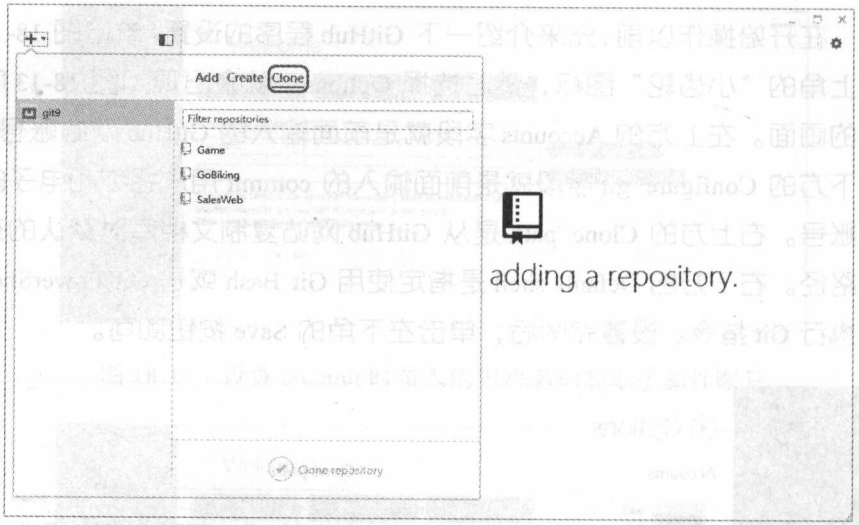


图 18-14　使用 GitHub 程序从 GitHub 网站复制 Git 文档库

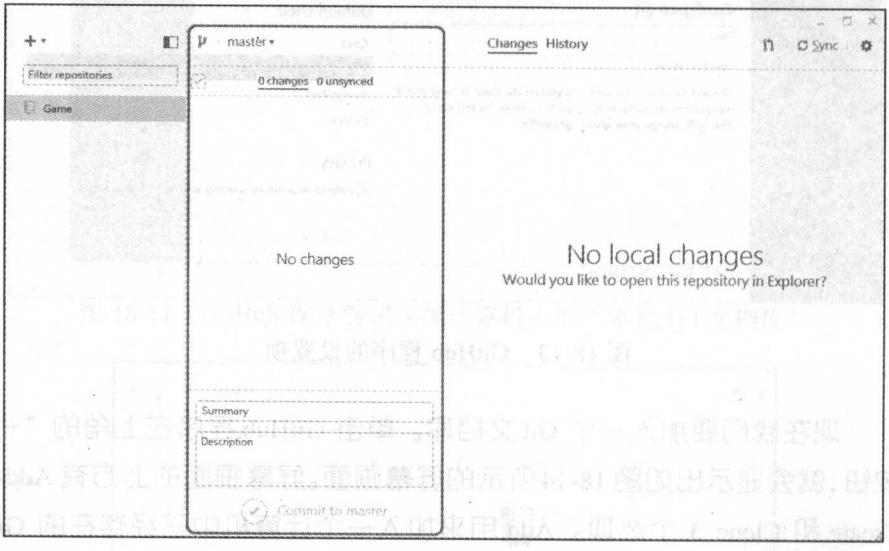


图 18-15　GitHub 程序管理 Git 文档库的屏幕画面

现在我们可以开始修改这个复制得到的程序项目，所有被修改的文件，都会显示在如图 18-15 所示的框选的区域内。接下来的操作方式如下：

1. 如果要执行 commit，先在下方的 Summary 字段输入操作说明，再单击最下面的 Commit 按钮。

2. 如果要新增分支，可以单击如图 18-15 所示的框选区域左上角的分支图标，然后输入新分支的名称，再单击 Create 按钮。创建新分支之后，会自动切换到这个新分支。

3. 分支图标右边显示的是当前正在操作中的分支名称。如果要切换分支，可以单击该分支名称，再从显示的分支列表中选择某一个分支，就会进行切换。

4. 如果要合并分支，先单击分支图标右边的分支名称，再单击右上角的 Manage，就会出现如图 18-16 所示的屏幕显示画面。假设我们要把 develop 分支合并到 master 分支。先用鼠标把如图 18-16 所示的屏幕画面中的 develop 分支，拖曳到下方最左边的 drag branch here 区域。然后再把 master 分支拖曳到下方第二个 drag branch here 区域。最后单击右边的 Merge 按钮。

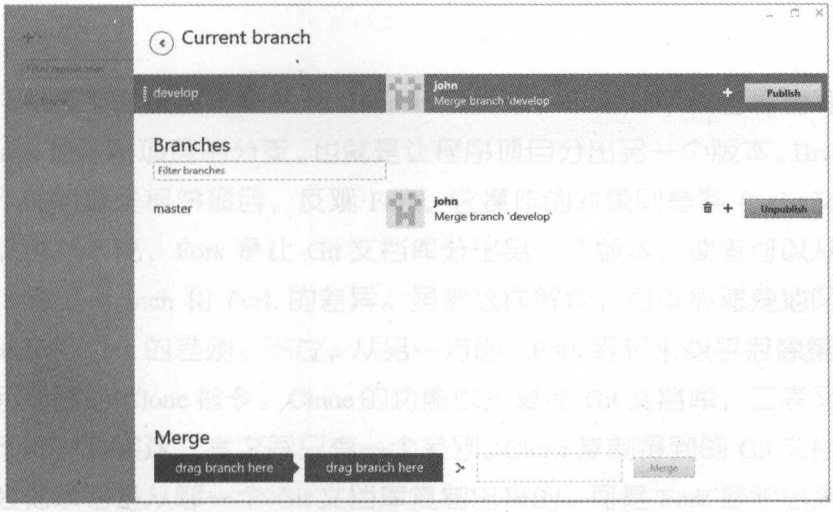


图 18-16　合并分支的操作界面

5. 如果当前操作中的分支有对应到 GitHub 网站上的 Git 文档库，GitHub 程序右上角则会显示 Sync 按钮。单击该按钮，就会把“本地 Git 文档库”的更新传送到 GitHub 网站上的 Git 文档库。如果当前操作中的分支还没有传送给 GitHub 网站上的 Git 文档库，GitHub 程序右上角则会显示 Publish 按钮。单击该按钮，就会把这个分支的数据传送到 GitHub 网站上的 Git 文档库。

6. 如果要删除分支，可以单击如图 18-16 所示的屏幕画面上的垃圾桶图标。但是要注意，删除分支会同时把“本地 Git 文档库”和 GitHub 网站上的 Git 文档库中的分支删除。

7. 如果想要执行 Git 指令，可以在如图 18-15 所示的屏幕画面右上角，单击“小齿轮”图标，然后选择 Open in Git Shell，就会启动 Git Bash 或是 Git PowerShell 程序。

了解了 GitHub 网站和程序的操作方法之后，下一个单元我们要介绍一种复制 Git 文档库的特殊方式，那就是 Fork。



第19课

Fork让Git文档库分家后再合并

从字义上来说，Fork 和 Branch 很类似，乍看之下很容易混淆。Branch 是程序项目的分支，也就是让程序项目分出另一个版本。Branch 操作的对象是程序项目，反观 Fork，它操作的对象则是整个 Git 文档库。也就是说，Fork 是让 Git 文档库分出另一个版本。读者可以从图 19-1 看出 Branch 和 Fork 的差异。虽然这样解释，可以很清楚地区分 Branch 和 Fork 的差别。不过，从另一方面，Fork 看起来似乎很像第 16 单元介绍的 Clone 指令。Clone 的功能也是复制 Git 文档库，二者又有何不同？其实这二者之间只有一个差别。Clone 复制得到的 Git 文档库不会记录它是从哪一个 Git 文档库复制出来的。可是 Fork 复制出来的 Git 文档库，会保留它的来源 Git 文档库。将来这个 Fork 得到的 Git 文档库可以再合并到原来的 Git 文档库。

补充说明

Fork 并不是 Git 程序原生的操作模式，它是网络上的 Git 服务网站自行衍生出来的一种新功能。

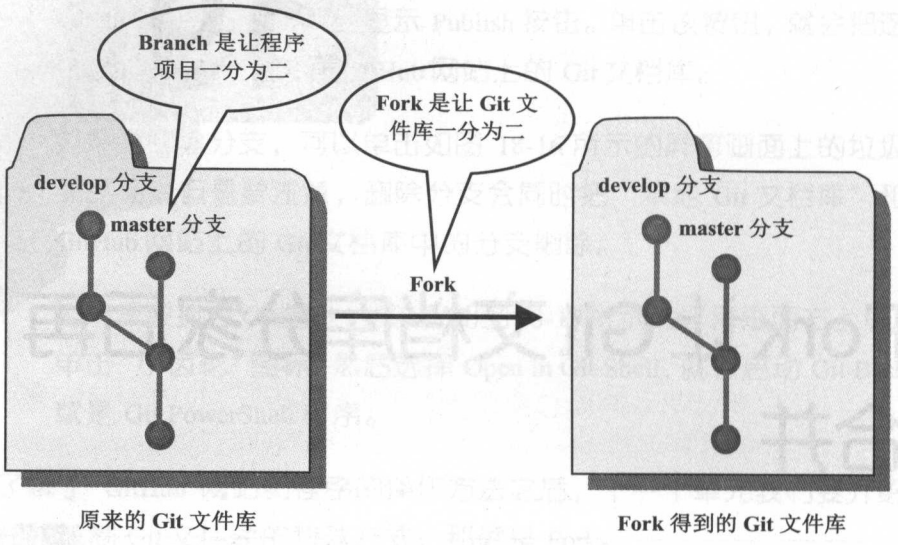


图 19-1　Fork 和 Branch 的差别

我们把图 19-1 再进行扩充，形成图 19-2，来说明 Fork 文档库一个完整的操作循环。首先我们使用 Fork，复制产生一个 Git 文档库。然后就如同操作一般 Git 文档库一样，在这个 Fork 得到的文档库上，执行 commit、创建分支、合并等各种操作。在同一段时间，原来的 Git 文档库也会进行修改和更新。我们可以在任意时间点，从原来的 Git 文档库获取更新，并且运用到我们 Fork 得到的文档库。这个操作就像是分支的 Rebase 指令一样，只不过它发生在 Git 文档库的层级。当我们的程序项目开发完成，并且 commit 到 Git 文档库之后，可以创建一个 Pull Request 信息，送给原来的 Git 文档库。原来 Git 文档库的拥有者，收到 Pull Request 的请求之后，如果决定采纳我们的成果，可以把我们的 Git 文档库的内容，合并到原来的 Git 文档库中。解释完

Fork 的原理和功能之后，接下来就来实际演练一下，亲身体验 Fork 的操作流程。

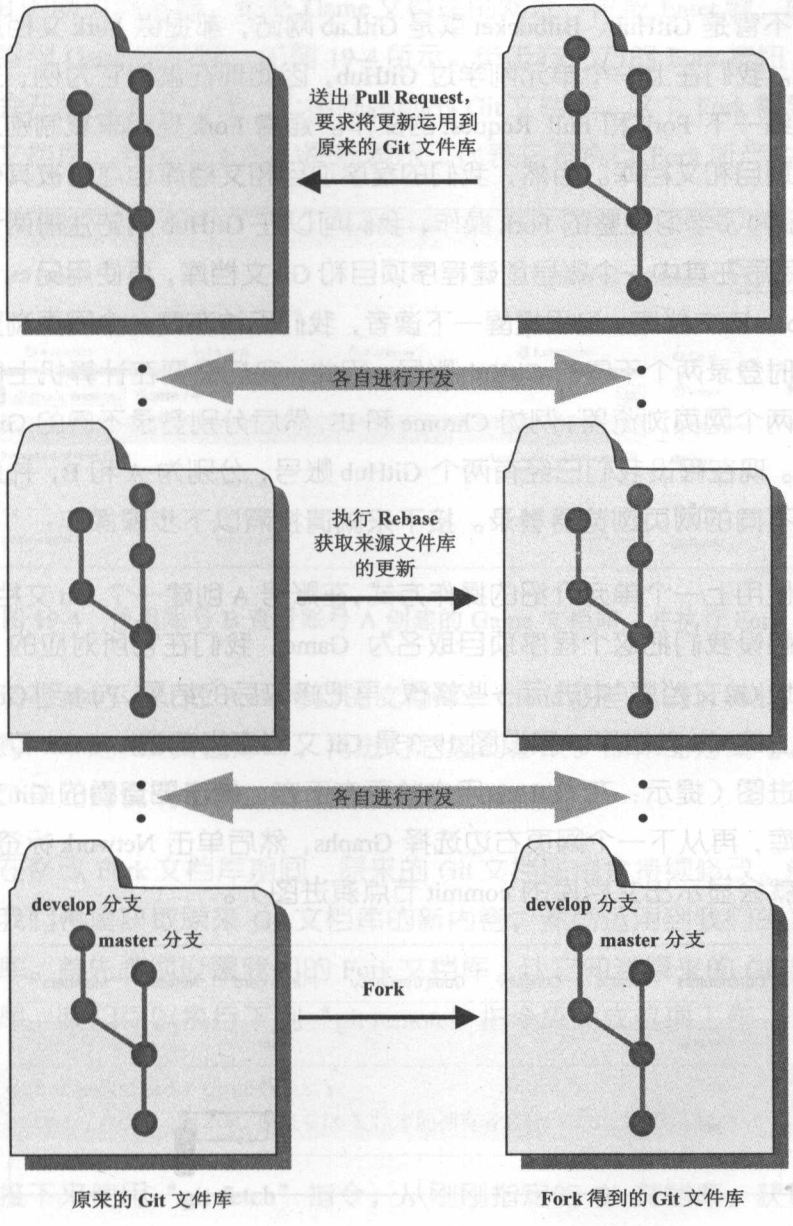


图 19-2　Fork 文档库之后的 Rebase 和 Pull Request 操作

19-1 Fork 和 Rebase Git 文档库

不管是 GitHub、Bitbucket 或是 GitLab 网站，都提供 Fork 文档库的功能。我们在上一个单元刚学过 GitHub，因此现在就以它为例，来实际演练一下 Fork 和 Pull Request 的操作。通常 Fork 是用来复制别人的程序项目和文档库。当然，我们的程序项目和文档库也可能被其他人 Fork。为了学习完整的 Fork 操作，我们可以在 GitHub 网站注册两个账号。然后在其中一个账号创建程序项目和 Git 文档库，再使用另一个账号 Fork 该文档库。这里提醒一下读者，我们无法在同一个网页浏览器上同时登录两个不同的 GitHub 账号。因此，我们需要在计算机上同时执行两个网页浏览器，例如 Chrome 和 IE，然后分别登录不同的 GitHub 账号。现在假设我们已经有两个 GitHub 账号，分别为 A 和 B，并且已经在不同的网页浏览器登录。接下来就请按照以下步骤操作：

1 步骤 使用上一个单元介绍的操作方式，在账号 A 创建一个 Git 文档库。假设我们把这个程序项目取名为 Game。我们在它所对应的“本地 Git 文档库”中进行一些修改，再把修改后的结果，Push 到 GitHub 上的 Git 文档库。假设图 19-3 是 Git 文档库当前的 commit 节点演进图（提示：在 GitHub 用户首页右下方，单击要查看的 Git 文档库，再从下一个网页右边选择 Graphs，然后单击 Network 标签页，就会显示出文档库的 commit 节点演进图）。

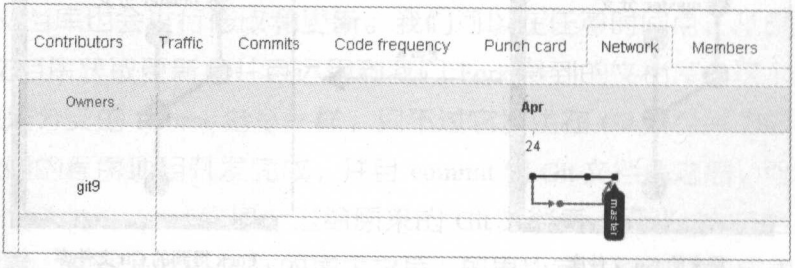


图 19-3　GitHub 账号 A 的 Game 文档库

2 步骤 接下来我们要使用账号 B 来 Fork 账号 A 创建的 Game 文档库。先在账号 A 的网页浏览器，复制 Game 文档库的网址。切换到账号 B 的网页浏览器，粘贴 Game 文档库的网址，再按 Enter 键，就会看到 Game 文档库，如图 19-4 所示。单击右上方的 Fork 按钮，就会在账号 B 中出现一个相同名称的 Git 文档库。这个 Fork 得到的文档库，会以分叉的小图标表示，代表它是执行 Fork 所产生。

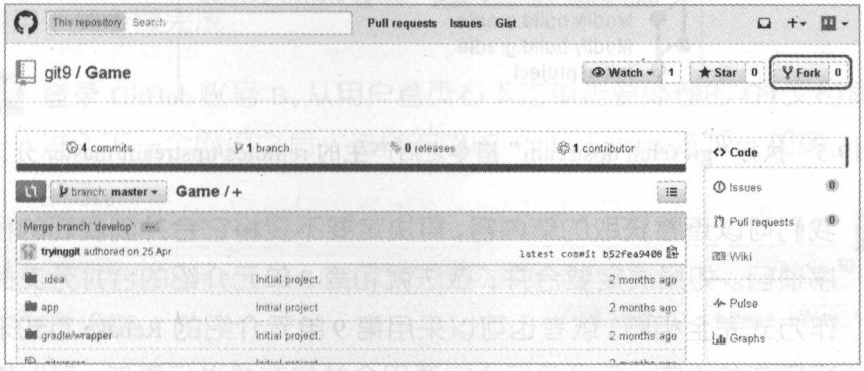


图 19-4　使用账号 B 查看账号 A 创建的 Game 文档库，并执行 Fork

3 步骤 现在我们可以把 Fork 得到的文档库，Clone 到自己的计算机上成为“本地 Git 文档库”，再进行后续的修改。操作的方式与上一个单元的说明相同。

4 步骤 在修改 Fork 文档库期间，原来的 Git 文档库也会持续修改。假设我们希望获取原来 Git 文档库的新内容，然后运用到我们的文档库。首先必须设置我们的 Fork 文档库，让它知道原来的 Git 文档库。我们可以执行下列“git remote”指令来完成这项工作：

```
git remote add upstream \
https://github.com/原来Git文档库的拥有者名称/Git文档库名称.git
```

5 步骤 接下来使用“git fetch”指令，从刚刚指定的 Git 文档库，获取新内容：

```
git fetch upstream
```

6 步骤 获取新内容之后，可以从 Git Bash 程序执行“gitk --all”指令，查看 Git 文档库中的分支。读者会发现多出一个 remotes/upstream/master 分支。这个分支就是从原来的 Git 文档库获取的新内容。

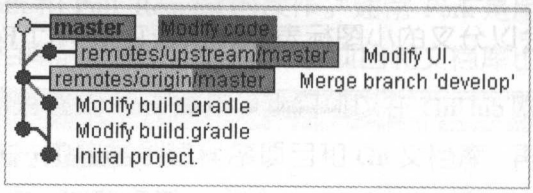


图 19-5　执行“git fetch upstream”指令之后产生的 remotes/upstream/master 分支

7 步骤 我们可以查看获取的新内容，再决定要不要将它合并到我们的程序项目。如果决定要合并，做法就和第 8 单元介绍的合并分支操作方式完全相同。或者也可以采用第 9 单元介绍的 Rebase 方式来运用新的内容。假设我们决定要用合并的方式进行更新，可以执行以下指令：

```
git checkout master
git merge upstream/master
```

如果决定要用 Rebase 的方式进行更新，则执行以下指令：

```
git checkout master
git rebase upstream/master
```

不管是合并或是 Rebase，都有可能出现冲突的情况。解决方式与第 8 单元和第 9 单元的说明相同。

如果后续要再从原来的 Git 文档库获取新的内容，只要重新执行步骤 5 到步骤 7 即可。以上就是 Fork 文档库和从原来的 Git 文档库获取更新的操作方式。接下来要介绍如何把修改的成果送回给原来的 Git 文档库。

19-2 创建 Pull Request 报告我们的更新

当我们在 Fork 的程序项目中完成开发工作之后，如果想要把成果合并到原来的 Git 文档库，可以通过 Pull Request 的方式，通知原来 Git 文档库的拥有者。以前面的例子来说，就是账号 A。现在假设我们要从账号 B，创建一个 Pull Request，送回给账号 A。这项操作可以使用下列步骤完成：

1 步骤 登录 GitHub 账号 B，从用户首页右下方单击要处理的 Git 文档库。进入下一个网页之后，选择右边的 Pull requests 选项，如图 19-6 所示。

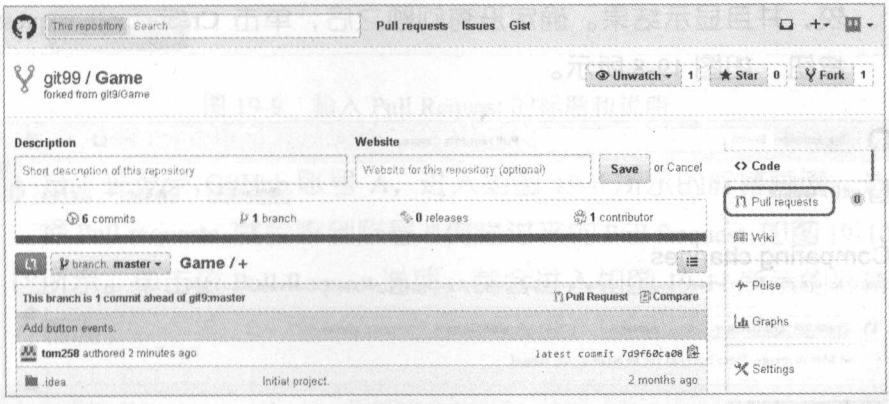


图 19-6 选择 Pull requests 准备通知原来 Git 文档库的拥有者

2 步骤 进入下一个网页屏幕画面后，单击 New pull request 按钮，如图 19-7 所示。

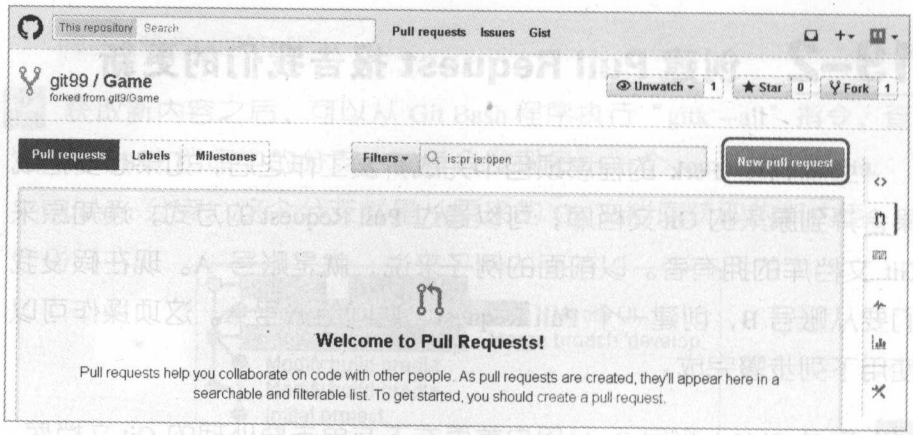


图 19-7　单击 New pull request 按钮

3 步骤 接下来的网页会将我们的 Git 文档库和原来的 Git 文档库进行比较，并且显示结果。确定没有问题之后，单击 Create pull request 按钮，如图 19-8 所示。

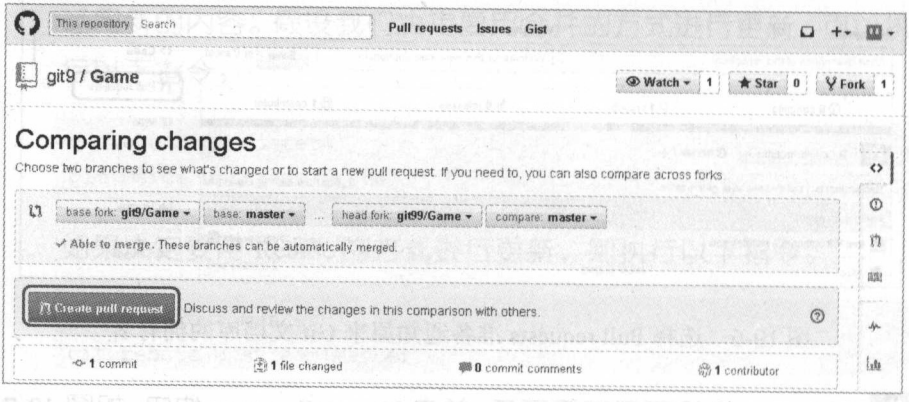


图 19-8　显示两个文档库的对比结果

4 步骤 最后一个网页需要输入 Pull Request 的标题和说明，输入完毕后，单击 Create pull request 按钮，如图 19-9 所示。



图 19-9　输入 Pull Request 的标题和说明

5 步骤　接下来登录 GitHub 账号 A，进入如图 19-6 所示的屏幕画面，选择 Pull requests，就会看到账号 B 传送过来的 Pull Request，如图 19-10 所示。单击该 Pull Request 选项，就会进入如图 19-11 所示的屏幕画面。

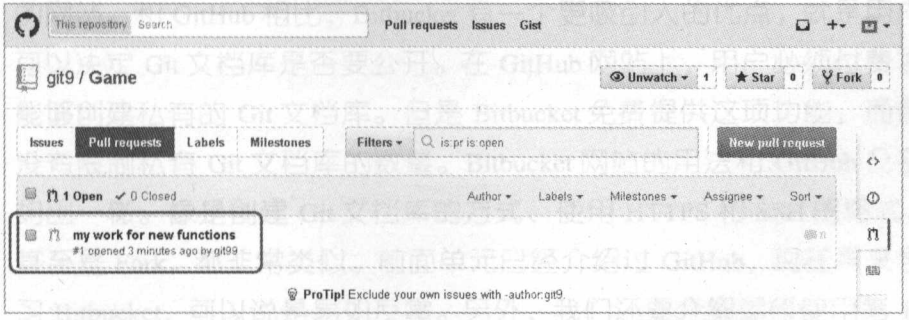


图 19-10　查看账号 B 传送的 Pull Request

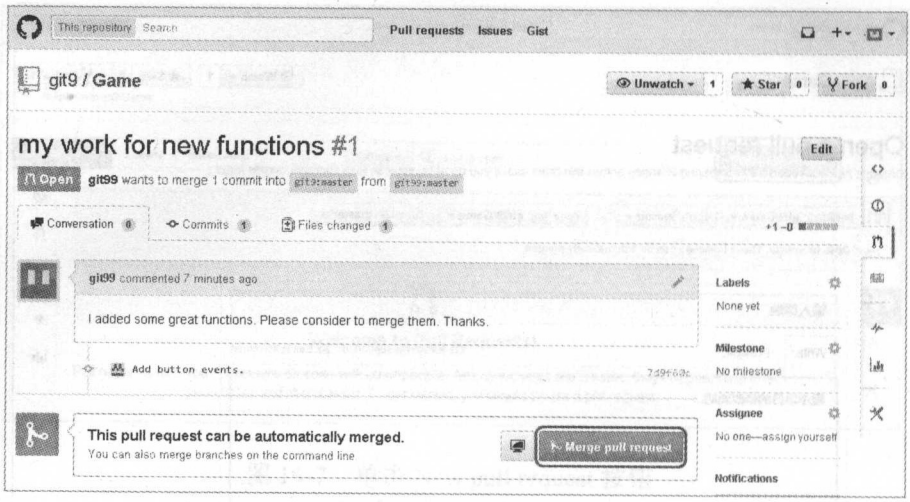


图 19-11　查看 Pull Request 的内容

6 步骤 我们可以在如图 19-11 所示的画面中查看 Pull Request 的内容，如果决定要将它合并到原来的 Git 文档库，可以单击 Merge pull request 按钮，然后在下一个屏幕画面单击 Confirm merge，就可以完成两个文档库的合并。



第20课

Bitbucket比GitHub更好用

Bitbucket网站和GitHub一样，都是专门提供存储Git文档库服务的网站。和GitHub相比，Bitbucket有一个更吸引人的优点，就是用户可以决定Git文档库是否要公开。在GitHub网站上，用户必须付费才能够创建私有的Git文档库。但是Bitbucket免费提供这项功能，而且没有限制私有Git文档库的数量。Bitbucket网站的用法和GitHub几乎如出一辙。像是创建Git文档库的方式、使用HTTPS和SSH的格式、甚至是Fork，都非常类似。前面单元已经介绍过GitHub，现在再来学习Bitbucket，可以说是易如反掌。另外，我们还要介绍如何使用第14单元的SourceTree软件来存取Bitbucket网站上的文档库。首先从注册成为Bitbucket网站的用户开始。

20-1 注册和设置 Bitbucket

首先必须找到 Bitbucket 官方网站，并且完成注册。开启网页浏览器，用百度或者 Google 搜索页搜索 Bitbucket，就可以找到它的官方网站。进入 Bitbucket 官方网站会看到如图 20-1 所示的画面。如果已经注册过，可以单击画面右上方的 Log in，切换到登录屏幕画面。如果是第一次使用，直接单击中间的 Get started for free，就会切换到如图 20-2 所示的画面。输入用户名称、密码和电子邮件账号，勾选 I'm not a robot，再单击下方的 Sign up，就会出现如图 20-3 所示的屏幕画面。现在我们要创建第一个 Git 文档库，选择中央右边的 Empty 按钮，在下方的 Name 字段输入 Git 文档库的名称。如果希望这个文档库是私有的，不对外公开，就勾选 This is a private repository，最后单击 Create 按钮。出现下一个屏幕画面后，单击 Done，就会显示如图 20-4 所示的屏幕画面。左边是菜单区，屏幕画面中央是操作说明。第一种方式是使用第 14 单元介绍的 SourceTree 程序，第二种方式是使用 Git 指令。我们可以进一步展开下方的子选项，其中有 Git 指令的范例。

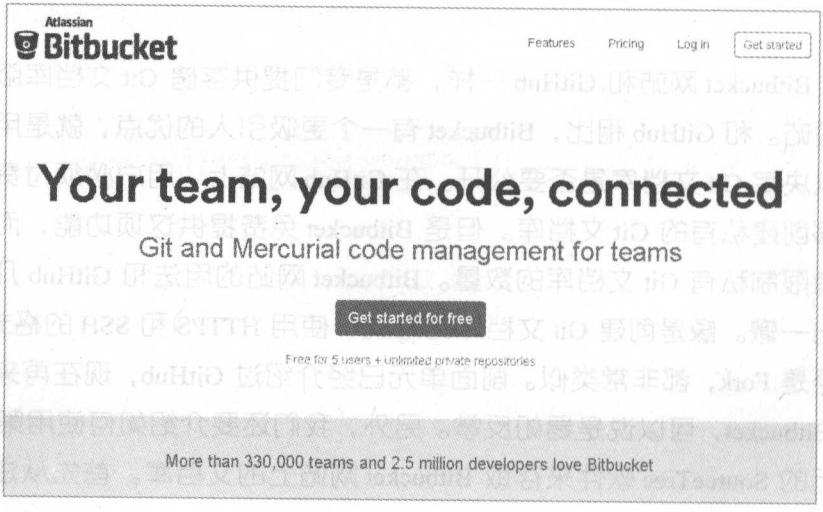


图 20-1　Bitbucket 网站首页



图 20-2　输入注册信息和选择使用模式

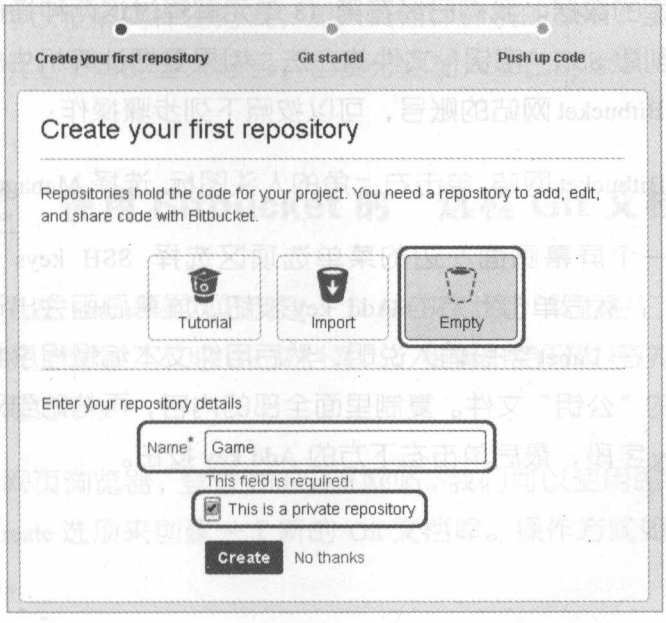


图 20-3　完成注册后的操作界面

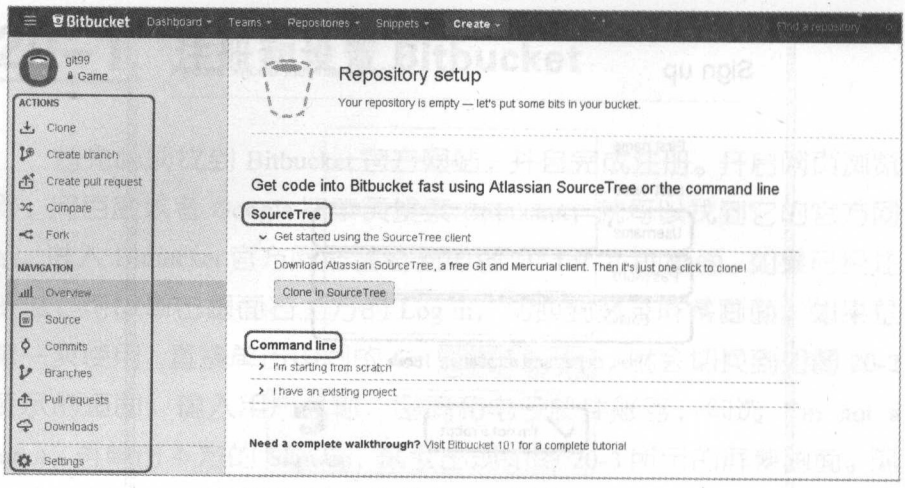


图 20-4　进入 Git 文档库后的操作界面

Bitbucket 和 GitHub 网站一样，里面的 Git 文档库都是属于“远程 Git 文档库”，而且同样可以使用 HTTPS 通信协议和 SSH 通信协议来传送文档库的数据。我们已经在第 18 单元解释过这两种通信协议的差别以及创建 SSH“密钥”文件的方法。如果要把计算机生成的“公钥”加入 Bitbucket 网站的账号，可以按照下列步骤操作：

1 步骤　登录 Bitbucket 网站，单击右上角的人头图标，选择 Manage account。

2 步骤　在下一个屏幕画面左边的菜单选项区选择 SSH keys（参考图 20-5），然后单击上方的 Add key 按钮，屏幕画面会出现一个对话框。在 Label 字段输入说明，然后用纯文本编辑程序打开计算机上的“公钥”文件。复制里面全部的内容，再粘贴到对话框中的 Key 字段，最后单击右下方的 Add key 按钮。

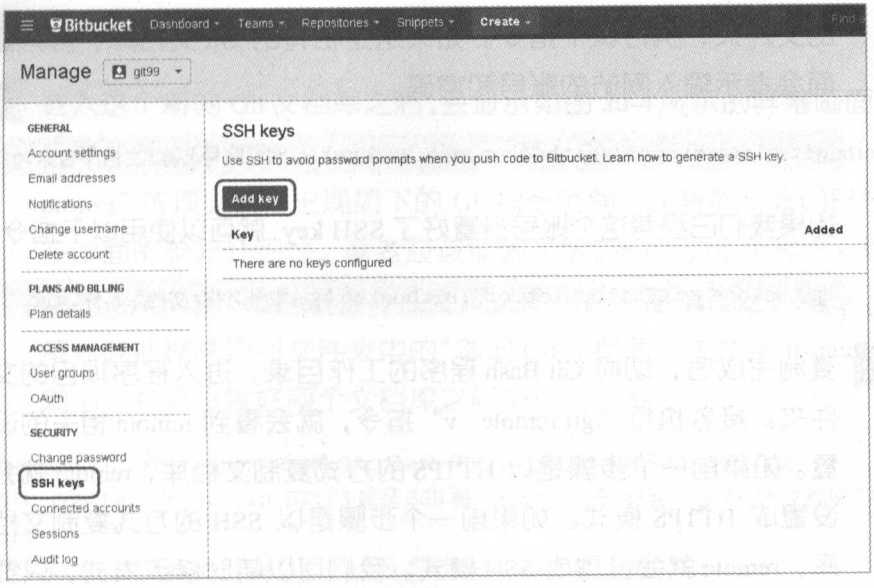


图 20-5　在 Bitbucket 网站加入我们计算机的“公钥”

我们可以按照同样的方式，加入其他计算机的 SSH“公钥”。同一台计算机上不同的登录账号，也要分别加入自己的 SSH“公钥”。

20–2　使用 Bitbucket 的“远程 Git 文档库”

假设先在 Bitbucket 网站创建一个 Git 文档库，然后将它复制到自己的计算机上成为“本地 Git 文档库”，接着开始开发程序项目，整个操作流程如下：

1 步骤　开启网页浏览器，登录 Bitbucket 网站。我们可以使用图 20-4 上方的 Create 选项来创建一个新的 Git 文档库。操作方式如同前面的说明。

2 步骤　现在我们要从 Bitbucket 网站复制 Git 文档库到自己的计算机中，成为“本地 Git 文档库”。启动 Git Bash 程序，切换到程序项目

的文件夹，执行以下指令。如果这是私有的 Git 文档库，屏幕画面会提示输入网站的账号和密码。

```
git clone https://bitbucket.org/Bitbucket 网站账号名称/文档库名称.git
```

如果我们已经帮这个账号设置好了 SSH key，就可以使用以下指令：

```
git clone git@bitbucket.org:Bitbucket 网站账号名称/文档库名称.git
```

3 步骤 复制完成后，切换 Git Bash 程序的工作目录，进入程序项目的文件夹。接着执行“git remote –v”指令，就会看到 remote 相关的设置。如果前一个步骤是以 HTTPS 的方式复制文档库，remote 就会设置成 HTTPS 模式。如果前一个步骤是以 SSH 的方式复制文档库，remote 就会设置成 SSH 模式。我们可以随时修改模式，只要使用“git remote set-url”指令即可：

```
git remote set-url origin HTTPS 或是 SSH 模式的 url
```

4 步骤 现在可以开始在这个程序项目的文件夹中创建程序项目，再使用前面学过的 Git 指令执行 commit、或是创建分支、执行合并等。如果要把“本地 Git 文档库”的更新送到 GitHub 网站，只要按照第 16 单元介绍的方法操作即可。

5 步骤 如果要删除 Bitbucket 网站上的文档库，可以选择图 20-4 左边菜单选项区的 Settings。再从显示的屏幕画面中选择 Delete repository，然后按照说明操作即可。

6 步骤 如果想要查看 Git 文档库的 commit 节点演进图，可以选择图 20-4 左边菜单选项区的 Commits。

第二种使用情况是我们先在计算机上创建“本地 Git 文档库”，再把它送到 Bitbucket 网站上的文档库进行存储，以下是完整的操作步骤：

1 步骤 先登录 Bitbucket 网站，创建一个新的 Git 文档库。

2 步骤 进入这个新的 Git 文档库之后，会显示如图 20-4 所示的屏幕画面。在屏幕画面中央的 Command line 下方，展开“I have an existing project”选项，就会出现如下的 Git 指令范例。这些指令我们已经在前面的单元介绍过，读者应该能够了解它们的功能（提示：可以参考本书附录的 Git 指令和使用说明）。按序执行这些指令，就可以把程序项目文件夹中的“本地 Git 文档库”，传送到 Bitbucket 网站，并且设置好两个文档库之间的对应关系。

```
cd /path/to/my/repo   ← 将 Git Bash 的工作目录切换到程序项目的文件夹
git remote add origin HTTPS 或是 SSH 模式的 url   ← 设置“远程 Git 文档库”的 url
git push -u origin -all   ← 上传“本地 Git 文档库”全部的分支
git push -u origin -tags ← 上传“本地 Git 文档库”分支中的标签
```

Bitbucket 网站还免费提供 SourceTree 程序，让我们可以用图形界面的方式操作 Git。我们已经在第 14 单元介绍过 SourceTree 程序的用法，这里我们再针对“远程 Git 文档库”的部分做进一步的说明。

20-3 使用 SourceTree 程序操作“远程 Git 文档库”

现在用前一个小节的两种操作情况为例，来介绍如何使用 SourceTree 程序，让“远程 Git 文档库”和“本地 Git 文档库”完成连接和更新。第一种情况是先在 Bitbucket 网站创建一个 Git 文档库，然后将它复制到自己的计算机中，成为“本地 Git 文档库”，接着开始开发程序项目，完整的操作流程如下：

1 步骤 开启网页浏览器，登录 Bitbucket 网站，使用图 20-4 上方的 Create 选项，来创建一个新的 Git 文档库。

2 步骤 启动 SourceTree 程序，单击工具栏最左边的 Clone/New 按钮（参考图 20-6），就会显示一个对话框。单击 Clone Repository 标签页，在 Source Path/URL 字段输入“远程 Git 文档库”的 URL。我们可以参考上一个小节的做法，输入 HTTPS 或是 SSH 格式的 URL。SourceTree 程序会视情况提示输入用户账号和密码。接着在 Destination Path 输入我们计算机上的程序项目文件夹路径，或是使用最右边的按钮，打开文件浏览对话框进行设置。完成之后，单击右下角的 Clone 按钮。SourceTree 程序会开始进行复制。

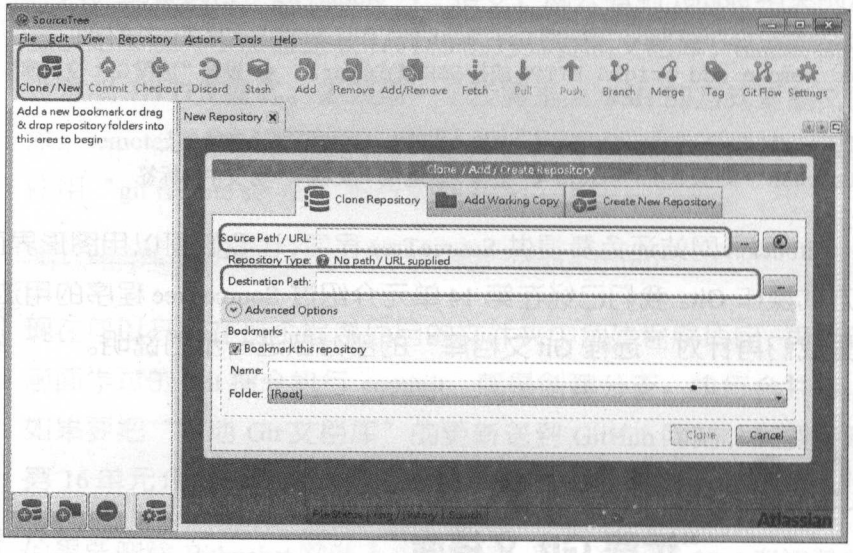


图 20-6　使用 SourceTree 程序从 Bitbucket 网站复制 Git 文档库

3 步骤 现在可以开始在这个程序项目的文件夹中创建程序项目，并且执行 commit、或创建分支、执行合并等等。如果要把“本地 Git 文档库”的更新送到 GitHub 网站，只要单击工具栏中的 Push 按钮，如图 20-7 所示。程序屏幕画面会出现一个对话框，在对话框中勾选要上传的分支，也可以在左下角勾选是否要上传分支中的标签。最后单击 OK 按钮。如果要从“远程 Git 文档库”下载更新，可以单击工具栏上的 Pull 按钮。

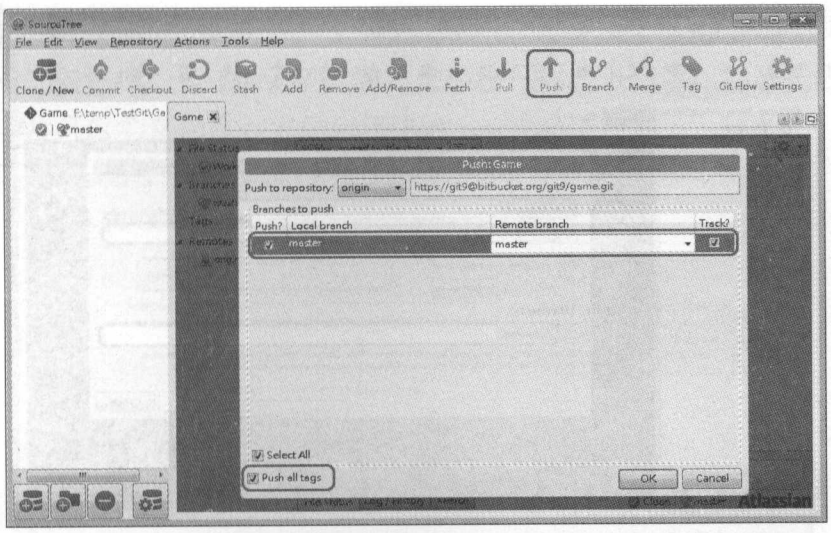


图 20-7　使用 SourceTree 程序将“本地 Git 文档库”的更新 Push 到“远程 Git 文档库”

补充说明

我们也可以使用 SourceTree 程序，从其他网站复制 Git 文档库到自己的计算机中，只要在步骤 2 中，换成设置其他网站的“远程 Git 文档库”的 URL 即可。

第二种使用情况是先在计算机上创建“本地 Git 文档库”，再把它送到 Bitbucket 网站上，以下是完整的操作流程：

1 步骤 登录 Bitbucket 网站，创建一个新的 Git 文档库。

2 步骤 按照前面介绍的方法，获取这个 Git 文档库的 URL。

3 步骤 切换到 SourceTree 程序，单击工具栏最右边的 Settings 按钮。屏幕画面会显示一个对话框，单击对话框左下方的 Add 按钮，就会出现如图 20-8 所示的屏幕画面。在 Remote name 字段输入“远程 Git 文档库”的名称，例如：origin。在 URL/Path 字段输入上一个步骤获取的“远程 Git 文档库”的 URL。在 Username 字段输入 Bitbucket 网站的账号，最后单击 OK 按钮。

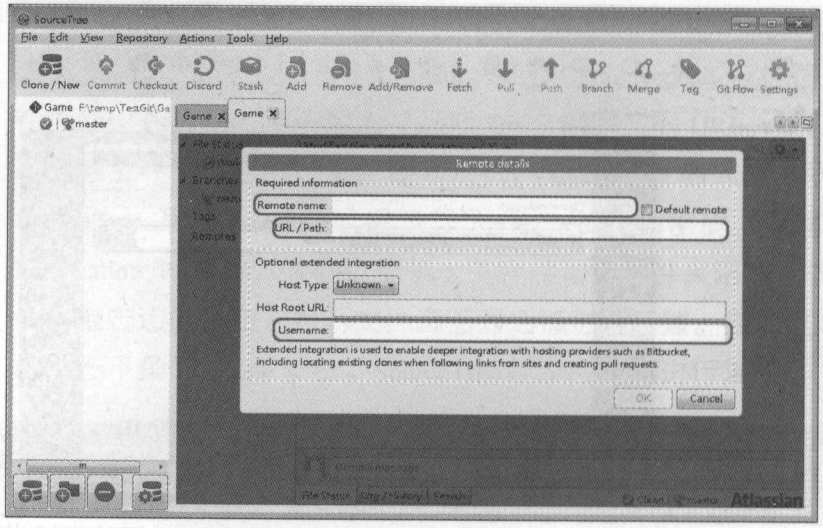


图 20-8　设置“本地 Git 文档库”对应的“远程 Git 文档库”的 URL

4 步骤 回到 SourceTree 程序的主屏幕画面，单击工具栏上的 Push 按钮，就会出现如图 20-7 所示的对话框。同样按照前面介绍的操作方式，就可以将“本地 Git 文档库”上传到 Bitbucket 网站。

最后我们介绍如何在 Bitbucket 网站上执行 Fork。

20–4　执行 Fork 和 Pull Request

Bitbucket 网站同样支持 Fork 和 Pull Request 的操作，而且流程和 GitHub 网站类似，以下是完整的操作流程：

1 步骤 首先是在 Bitbucket 网站上找到要 Fork 的文档库，并且复制该文档库的网址。我们只能 Fork 别人公开的文档库或是自己的文档库。

2 步骤 登录 Bitbucket 网站，把要 Fork 的文档库的网址粘贴到浏览器的网址栏，再按 Enter 键，就会连接到该文档库。

3 步骤 在屏幕画面左边的菜单选项区选择 Fork，就会显示如图 20-9 所示的对话框。在 Name 字段输入文档库名称，下方可以勾选是否要公开这个文档库，然后单击 Fork repository。

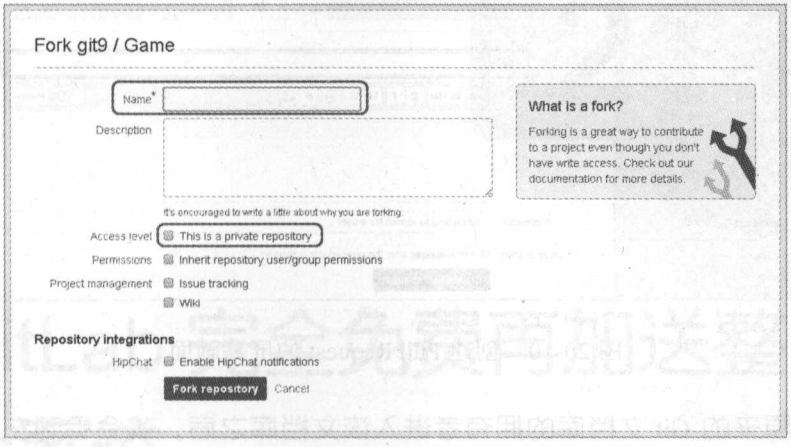


图 20-9　Fork 功能的操作界面

4 步骤 接下来就可以将这个 Git 文档库复制到自己的计算机中，成为“本地 Git 文档库”，并且进行修改。

5 步骤 如果 Bitbucket 网站上原来的 Git 文档库又被修改了，我们想要将修改运用到自己 Fork 的文档库。可以参考上一个单元的做法，使用“git fetch”指令将新的修改下载到“本地 Git 文档库”，然后进行合并。最后把合并后的结果上传到 Bitbucket 网站上的文档库。

6 步骤 当我们在 Fork 的文档库中完成开发工作之后，如果想要把成果合并到原来的 Git 文档库，可以在 Bitbucket 网站上进入 Fork 的文档库，在左边的菜单选项区选择 Create pull request，就会显示如图 20-10 所示的画面。在 Title 字段输入标题，如果想要加入更详细的说明，可以填写在下方的 Description 字段中。完成后单击 Create pull request 按钮。

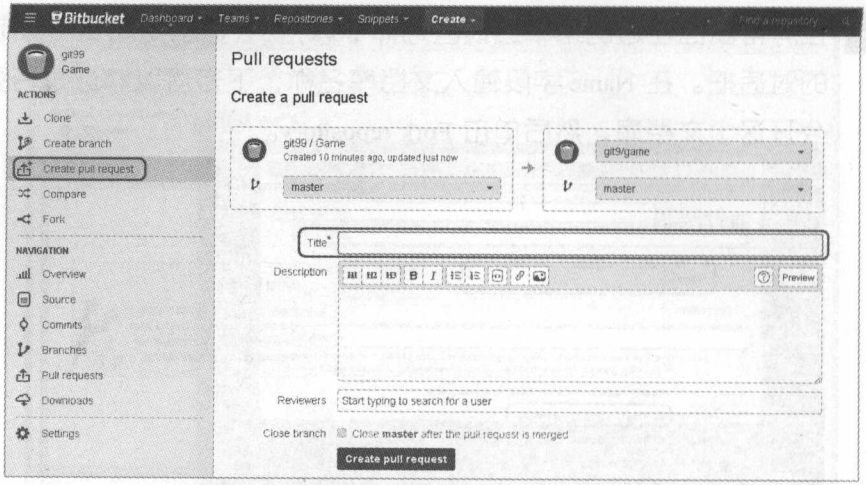


图 20-10　创建 Pull Request 的屏幕画面

7 STEP 原来的 Git 文档库的拥有者进入该文档库之后，就会看到左边菜单选项区下方的 Pull requests 选项后面显示了一个数字，表示有人传送了 Pull Request。选择该选项就会在屏幕画面上列出 Pull Request 的列表。单击里面的 Pull Request 选项，就会显示如图 20-11 所示的屏幕画面，里面会列出相关的信息。如果决定要采纳这些修改，可以单击右上方的 Merge 按钮。如果决定不要采用，则单击 Decline 按钮。



图 20-11　查看传送过来的 Pull Request



第21课

GitLab 完全免费再加送整个系统

Bitbucket 网站已经比 GitHub 大方得多，让我们可以免费创建私有的 Git 文档库。但是 GitLab 网站又更进一步，其他网站要收费的项目，GitLab 完全免费提供。不仅如此，它还把整个系统的程序代码公开，让大家可以在自己的计算机上安装，创建完全私有的 GitLab 网站。这个单元我们就来介绍 GitLab 网站的用法，以及如何在自己的计算机上创建一个 GitLab 网站。

GitHub 和 Bitbucket 网站提供的功能，在 GitLab 网站上一样都不缺。操作的方式和流程也很类似。要使用 GitLab 网站的服务之前，同样要先完成注册，并且到设置账号时填写的电子邮件地址对应的电子邮箱中收取认证邮件，单击其中的认证链接。注册完成后，登录 GitLab 网站，就会显示如图 21-1 所示的画面。单击 New Project 按钮，就会切

换到如图 21-2 所示的屏幕画面。在 Project path 字段输入文档库的名称，下方的 Visibility Level 可以设置这个 Git 文档库是完全私有、或可以被 GitLab 网站的登录账号看见、或完全公开。最后单击 Create project 按钮。

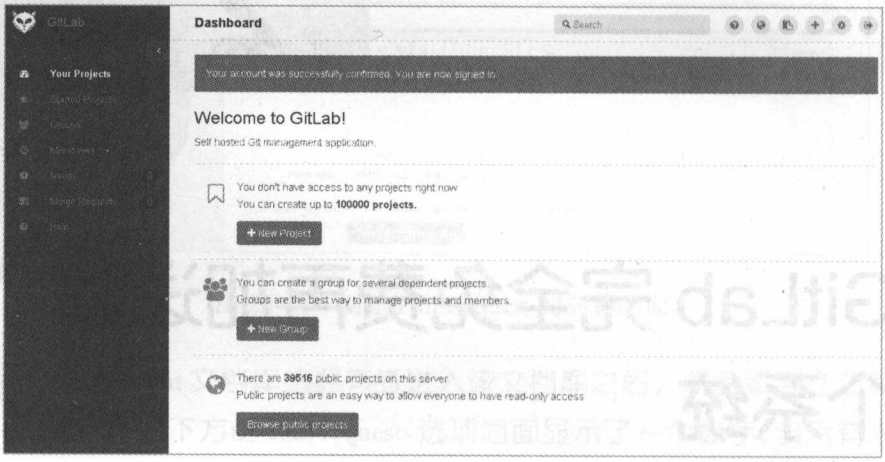


图 21-1　第一次登录 GitLab 网站时屏幕显示的画面

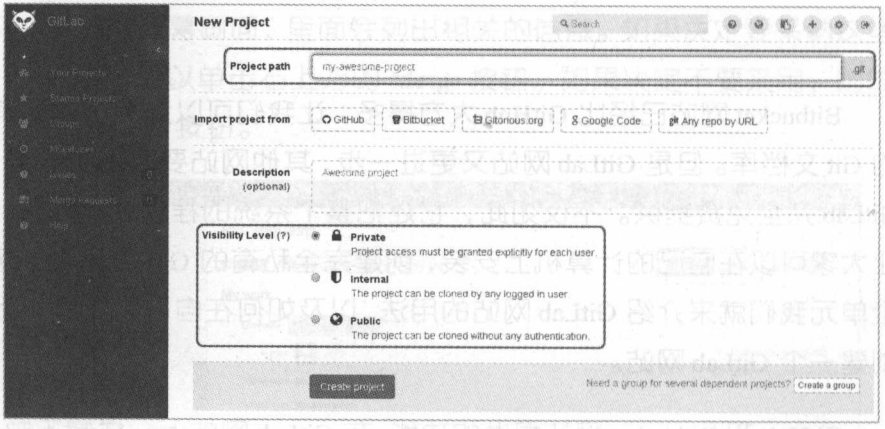


图 21-2　创建新的 Git 文档库

创建新的 Git 文档库之后，会进入如图 21-3 所示的画面，其中包含了使用说明。在网页右下角有一个 Remove project 按钮，这个按钮可以用来删除这个 Git 文档库。GitLab 同样提供了 HTTPS 和 SSH 两种

传输数据的方式。如果要使用 SSH，必须先在自己的计算机上生成一组“公钥”。详细的操作步骤可以参考第 18 课的说明。生成“公钥”之后，回到 GitLab 网站，单击屏幕画面右上角的 Profile settings 按钮（小齿轮图标）。单击网页左边菜单选项区中的 SSH Keys 选项，就会出现如图 21-4 所示的屏幕画面。选择右上角的 Add SSH Key，就会显示下一个屏幕画面。在 Title 字段输入说明，然后用纯文本编辑程序打开我们计算机上的“公钥”文件。复制里面的全部内容，粘贴到 GitLab 网站中的 Key 字段，最后单击 Add key 按钮。

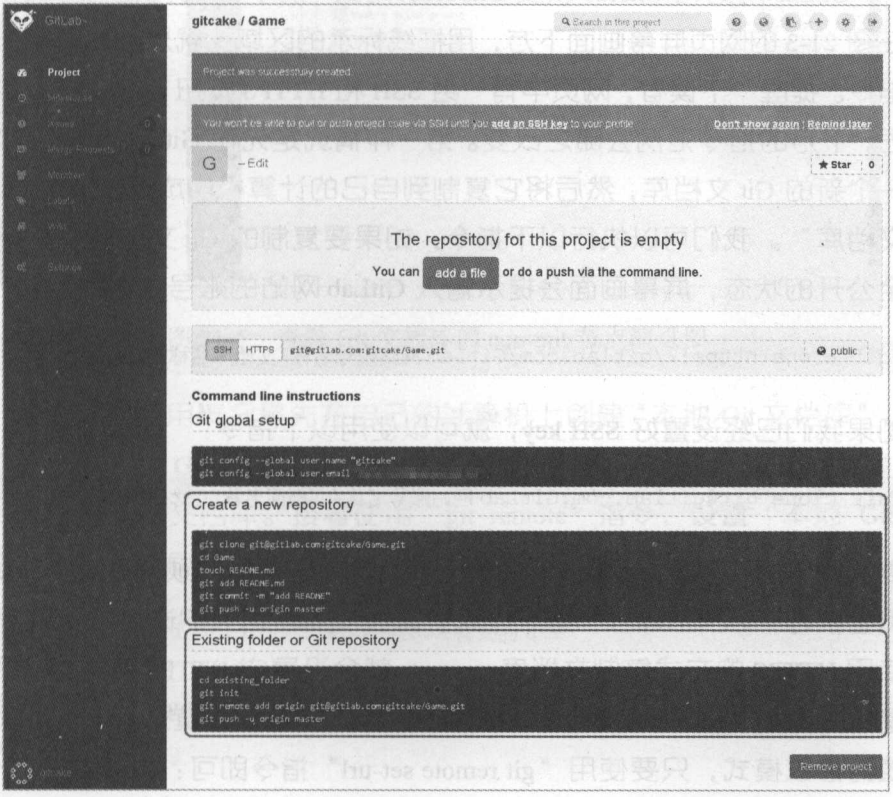


图 21-3　创建 Git 文档库之后显示的操作说明



图 21-4　在设置 SSH Keys 时屏幕显示的画面

GitLab 网站和 GitHub 以及 Bitbucket 网站一样，有两种使用情况。在图 21-3 的网页屏幕画面下方，用框线标示的区域，就是相关的操作指令。提醒一下读者，网页中有一组 SSH 和 HTTPS 按钮，单击它们之后，下方的指令范例会随之改变。第一种情况是先在 GitLab 网站创建一个新的 Git 文档库，然后将它复制到自己的计算机中成为“本地 Git 文档库”。我们可以执行以下指令，如果要复制的 Git 文档库不是完全公开的状态，屏幕画面会提示输入 GitLab 网站的账号和密码。

```
git clone https://gitlab.com/GitLab 网站账号名称/文档库名称.git
```

如果我们已经设置好 SSH key，就可以使用以下指令：

```
git clone git@gitlab.com:GitLab 网站账号名称/文档库名称.git
```

复制完成后，切换 Git Bash 程序的工作目录，进入程序项目的文件夹，然后执行“git remote -v”指令，就会看到 remote 相关的设置。如果是使用 HTTPS 的方式复制文档库，remote 就会设置成 HTTPS 模式。如果是以 SSH 的方式复制文档库，remote 就会设置成 SSH 模式。我们可以随时修改模式，只要使用“git remote set-url”指令即可：

```
git remote set-url origin HTTPS 或是 SSH 模式的 url
```

现在可以开始在这个文件夹中创建程序项目，再使用前面学过的 Git 指令执行 commit、或创建分支、执行合并等。如果要把“本地 Git

文档库”的更新送到 GitLab 网站，只要按照第 16 课介绍的方法操作即可。如果想要在 GitLab 网站查看 Git 文档库的 commit 节点演进图，可以先进入 Git 文档库的操作界面，然后在左边的菜单选项区选择 Network，就会显示如图 21-5 所示的结果。如果想要查看 commit 的统计图，可以从菜单选项区选择 Graphs。

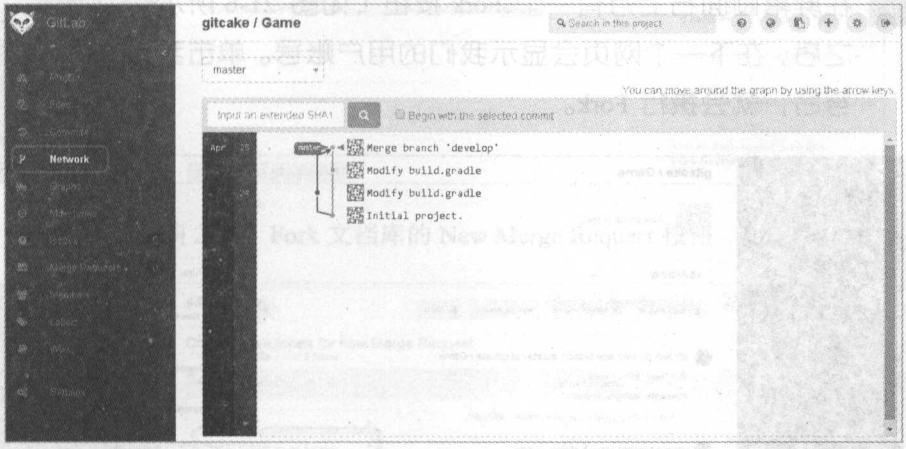


图 21-5　查看 Git 文档库的 commit 节点演进图

第二种使用情况是先在自己的计算机上创建“本地 Git 文档库”，再把它传送到 GitLab 网站。这种方式还是要先登录 GitLab 网站，创建一个新的 Git 文档库。接着使用“git remote”指令，设置“本地 Git 文档库”所对应的 GitLab 网站上的文档库：

```
git remote add origin HTTPS 或是 SSH 模式的 url
```

最后再使用“git push”指令，将“本地 Git 文档库”传送到 GitLab 网站上的文档库：

```
git push -u origin master
```

GitLab 网站也支持 Fork 和 Pull Request 的功能，只不过 GitLab 网站把 Pull Request 叫作 Merge Request，以下是完整的操作流程：

1 步骤 在 GitLab 网站上找到要 Fork 的文档库，复制该文档库的网址。我们只能 Fork 别人公开的文档库或是自己的文档库。

2 步骤 登录 GitLab 网站，把要 Fork 的文档库的网址粘贴到浏览器的网址栏，再按 Enter 键，就会连接到该文档库。

3 步骤 在屏幕画面右上方有一个 Fork 按钮（如图 21-6 所示）。单击它之后，在下一个网页会显示我们的用户账号。单击我们的用户账号后，就会执行 Fork。



图 21-6　执行 Fork 功能的操作界面

4 步骤 接下来就可以将这个 Git 文档库复制到自己的计算机中，成为“本地 Git 文档库”，并且进行修改。

5 步骤 如果 GitLab 网站上原来的 Git 文档库又被修改了，我们想要将修改运用到 Fork 的文档库，则可以按照第 19 课的做法，使用“git fetch”指令将新的修改下载到“本地 Git 文档库”，然后进行合并。最后把合并后的结果上传到 GitLab 网站上的 Fork 文档库。

6 步骤 当我们在 Fork 的文档库中完成开发工作之后，如果想要把成果合并到原来的 Git 文档库，可以从 GitLab 网站上进入 Fork 的文档库，在屏幕画面右边有一个 New Merge Request 的按钮（参考图 21-7），单击它之后会显示如图 21-8 所示的屏幕画面。



图 21-7　Fork 文档库的 New Merge Request 按钮

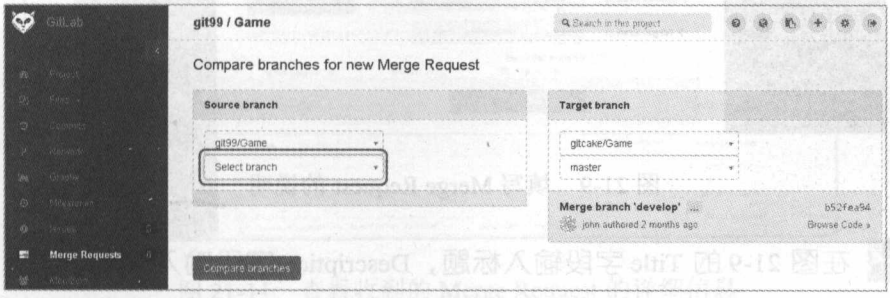


图 21-8　在设置 Merge Request 时屏幕显示的画面

7 步骤 如图 21-8 所示的屏幕画面会显示我们 Fork 的文档库以及原来的文档库。我们必须选择要将哪一个分支合并到原来的文档库。设置好分支之后，单击 Compare branches 按钮，就会显示如图 21-9 所示的屏幕画面。

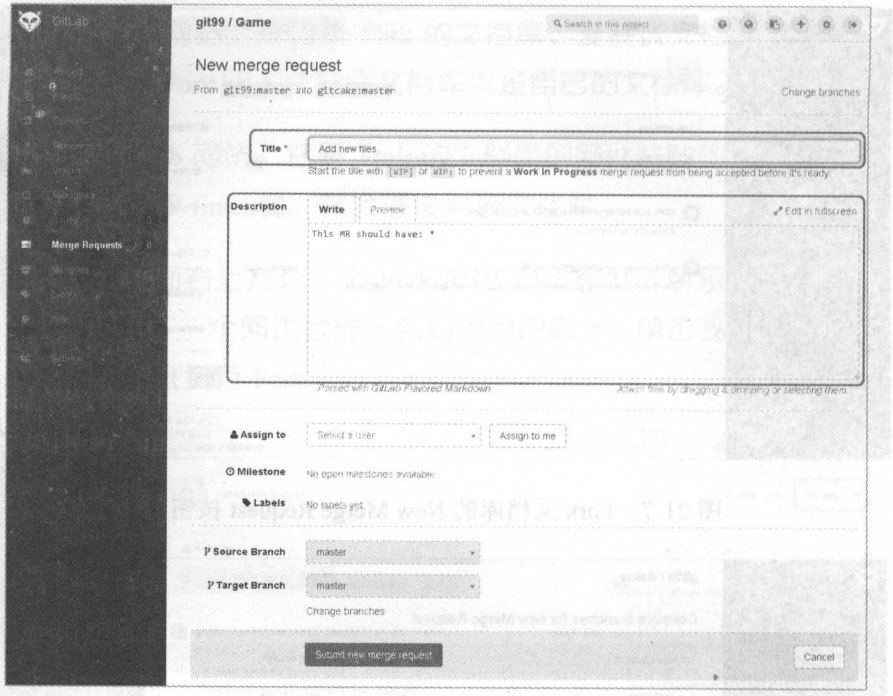


图 21-9　填写 Merge Request 的说明

8 步骤 在图 21-9 的 Title 字段输入标题，Description 字段输入说明，再单击 Submit new merge request 按钮，这个 Merge Request 就会送到对方的账号。

9 步骤 当对方进入这个文档库时，会在左边的菜单选项区显示出有未处理的 Merge Request，如图 21-10 所示。单击该选项，就会显示 Merge Request 的列表。单击列表上的选项，就会显示 Merge Request 详细的信息，如图 21-11 所示。查看信息之后，如果决定要执行合并，单击 Accept Merge Request 按钮即可。

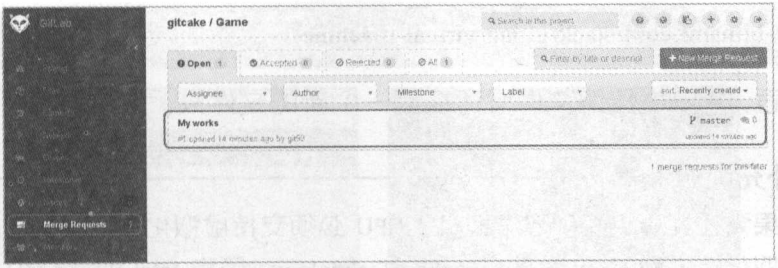


图 21-10　在收到 Merge Request 时屏幕显示的画面



图 21-11　查看收到的 Merge Request 的详细信息

21-1　自己架设 GitLab 网站

GitLab 除了免费提供所有的服务以外，它还开放网站的程序代码，让我们可以自己下载安装。但是 GitLab 系统只能够安装在 Linux 平台，如果是 Windows 系统，必须先安装虚拟机软件，然后创建一个虚拟机，再安装 Linux 操作系统，最后再安装 GitLab。如果真的按照这种方式来做，需要耗费不少时间。当然如果读者想要尝试试看，可以按部就班地实际操作一次。但是如果想要尽快创建一个自己的 GitLab 网站，还有一个非常简单的做法。我们可以到以下网站，下载它们已经创建好的 Linux 虚拟机，里面就已经架设好了 GitLab 系统：

https://bitnami.com/stack/gitlab/virtual-machine

下载后，将 ZIP 压缩文件解压缩到一个新的文件夹。

补充说明

如果要在计算机上创建虚拟机，CPU 必须支持虚拟化技术。Intel 的虚拟化技术称为 Intel Virtualization Technology，简称 Intel VT。AMD 的虚拟化技术称为 AMD Virtualization，简称 AMD-V。我们必须在计算机的 BIOS 设置中启用 CPU 的虚拟机功能，才能够使用虚拟机软件创建虚拟机。

接下来，我们只需要用一个虚拟机软件来启动这个虚拟机即可。VMware Player Pro 软件提供非商业用途的免费使用版权，我们可以用它来启动虚拟机。首先用百度或者 Google 来找到它的官方网站，下载安装程序并完成安装。安装完毕之后，启动 VMware Player 程序，就会显示如图 21-12 所示的画面。选择 Open a Virtual Machine，找到虚拟机文件将它打开，就会出现如图 21-13 所示的屏幕画面。我们可以使用下方的 Edit virtual machine settings 查看虚拟机的设置，并且加以修改。例如增加虚拟机内存的容量，让它执行起来比较顺畅。设置完毕之后，选择 Play virtual machine，就会开始启动虚拟机。

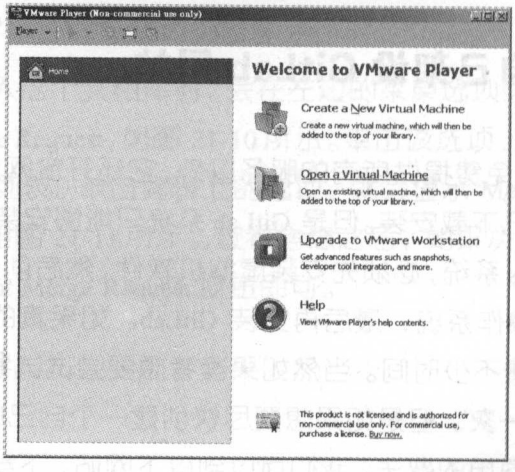


图 21-12　VMware Player 程序的启动画面

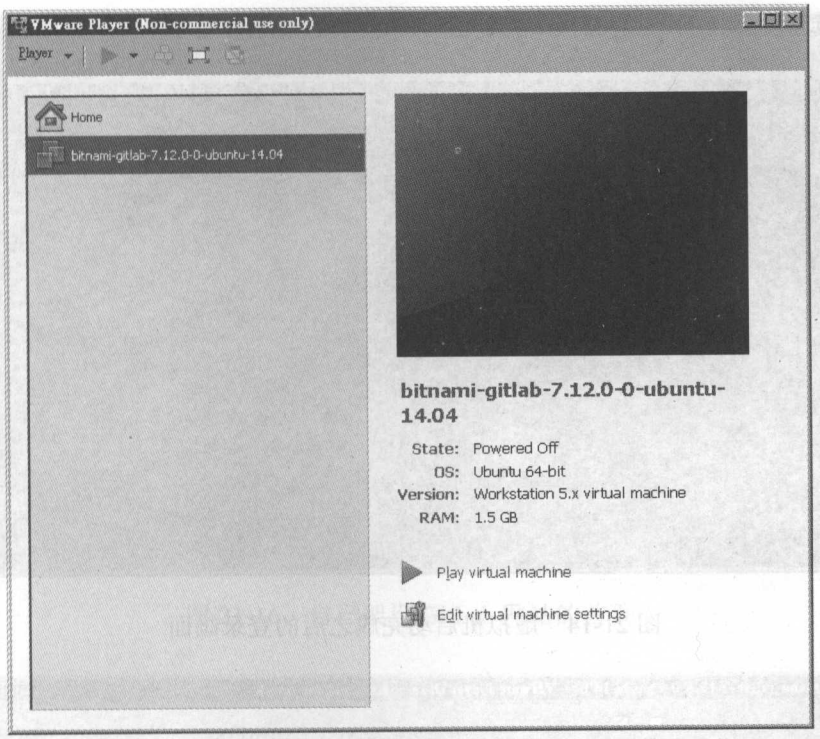


图 21-13　虚拟机文件的操作界面

虚拟机启动完成之后，会出现如图 21-14 所示的屏幕显示画面。它就是一个 Linux 操作系统，我们可以使用默认账号 bitnami 和默认密码 bitnami 登录，然后执行 ifconfig 指令，查看它的 ip 地址，如图 21-15 所示。现在我们可以开启计算机上的网页浏览器，在网址栏输入这个 ip 地址，就会出现 GitLab 网站的首页，如图 21-16 所示。如果读者想要查询 GitLab 网站安装的相关信息，可以连接到以下网址：

https://about.gitlab.com/installation/

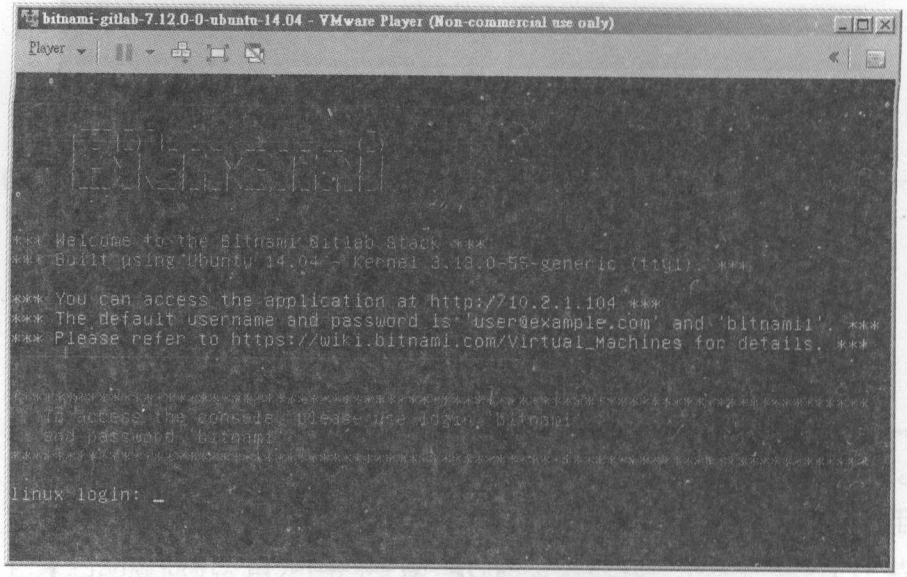


图 21-14　虚拟机启动完成之后的登录画面

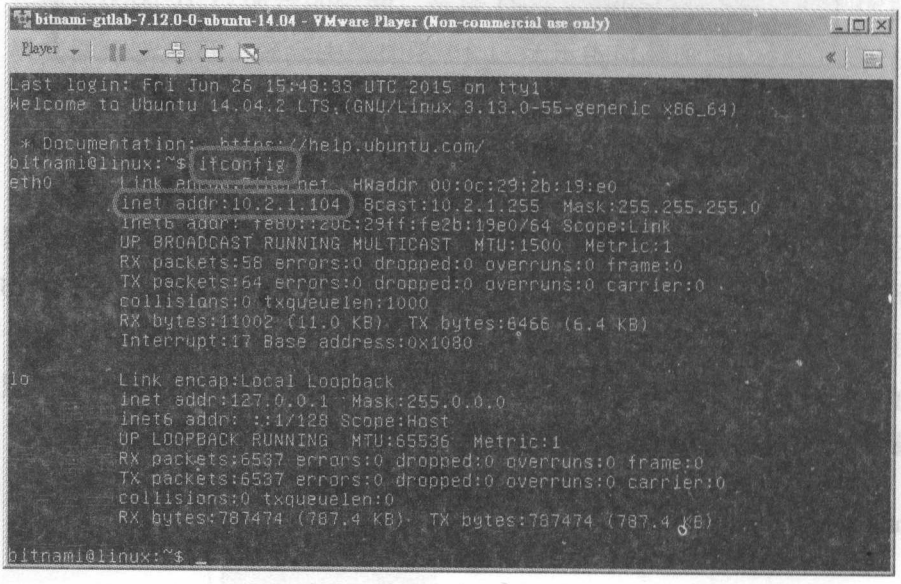


图 21-15　执行 ifconfig 指令查看 GitLab 使用的 ip 地址



图 21-16　自己架设 GitLab 网站的首页

虽然 GitLab 网站提供了非常多的功能，但是一般软件公司对于程序项目的管理，并不需要这么复杂的操作。一般软件公司的“远程 Git 文档库”放在服务器上，由专门负责的人员进行管理。开发人员的权限设置也是由项目负责人管理和控制的。大部分的开发人员并不需要自己创建程序项目。因此对于一般公司来说，并不需要架设自己的 GitLab 网站，比较合适的做法是使用 HTTP 服务器，搭配 Git 程序，或是 SSH 服务器加上 Git 程序，甚至也可以使用简单的网络文件夹共享机制，就可以创建“远程 Git 文档库”，这样在使用上更加方便。从下一个单元开始，我们将介绍几种自行创建 Git Server 的方式，让读者可以按照自己的网络架构和需求，选择最合适的方法。

第五部分　架设 Git Server

第 22 课　使用 Windows 共享文件夹或是 Git Daemon 作为 Git Server

第 23 课　使用 HTTP/HTTPS 访问 Git Server

第 24 课　使用 SSH 认证和加密的 Git Server



第22课

使用 Windows 共享文件夹或是 Git Daemon 作为 Git Server

这个单元先介绍两种访问远程 Git 文档库最简单的做法。这两种方法都不需要安装任何其他软件。第一种是使用 Windows 操作系统本身的网络“共享文件夹”功能；第二种是使用 Git 内建的 daemon 功能。

22-1 把 Windows“共享文件夹”作为“远程 Git 文档库”

如果 Git Server 是在局域网络中运行，而且每一位参与项目开发的人员都能够使用 Windows 的“共享文件夹”，那么最简单的做法就是直接创建一个“共享文件夹”，用它来存储远程 Git 文档库，如图 22-1 所示。做法如下：

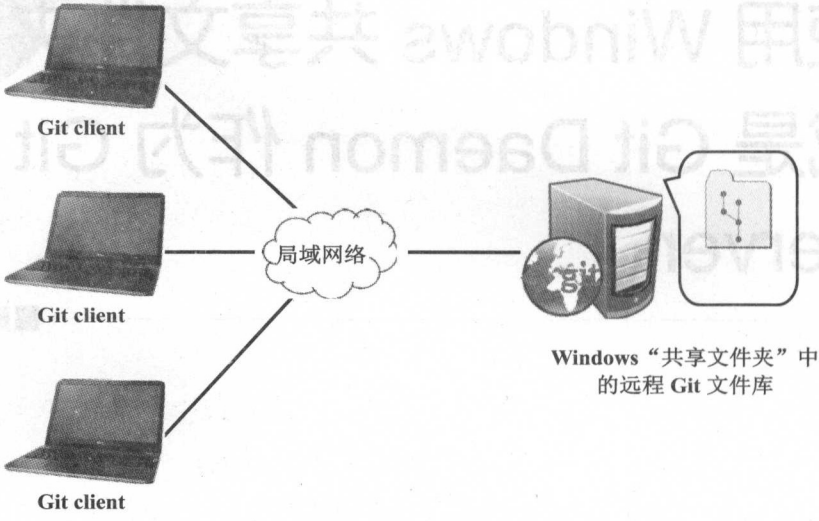


图 22-1　使用 Windows“共享文件夹”存储远程 Git 文档库

1 步骤 在要作为 Git Server 的计算机上，按照第 1 单元介绍的方式，完成 Git 程序的安装。

2 步骤 在硬盘上创建一个新的文件夹，这个文件夹将被设置为“共享文件夹”，里面会存储 Git 远程文档库，可以将它取名为 GitRemoteRepo。

3 步骤 启动 Windows 文件资源管理器，用鼠标右键单击这个新的文件夹。从快捷菜单中选择设置“共享”的选项将此文件夹设置为

共享。我们可以设置哪些用户能够存取，还可以进一步设置读或写的权限。

4 步骤 启动 Git Bash 程序，切换到这个文件夹，执行下列指令，创建一个远程 Git 文档库：

```
git init --bare (程序项目名称)
```

5 步骤 到任何一台其他的计算机中启动 Git Bash，切换到存储程序项目的文件夹。执行以下指令，从刚刚创建的远程 Git 文档库复制程序项目。其中的 GitRemoteRepo 是共享文件夹的名称：

```
git clone //服务器名称/GitRemoteRepo/程序项目名称
```

6 步骤 现在可以开始修改复制的程序项目，并且把文件加入 Git 文档库中。如果要将更新后的 Git 文档库送回给远程 Git 文档库，可以执行以下指令：

```
git push -u origin master
```

这种做法的好处是可以独立控制每一个用户的读写权限。如果有多个程序项目，而且每一个程序项目的用户和读写权限都不相同。我们可以为每一个程序项目创建专用的“共享文件夹”，独立管理和控制每一个程序项目的用户和权限。

22-2 使用 Git Daemon 创建 Git Server

Git Daemon 是 Git 内建的网络访问服务。使用 Git Daemon 不需要安装其他程序，只要使用一般的 Git 指令就可以启动。但是缺点是它没有授权的管理和控制机制。也就是说，只要知道链接的 URL，就可以自由访问。不过我们可以设置哪些程序项目要开放给外部访问，哪些不开放。Git Daemon 使用的是专用的通信协议（称为 Git protocol）和

9418 号通信信口（我们也可以指定其他的通信信口）。现在假设计算机的 D 磁盘中，有一个 GitRemoteRepo 文件夹，里面有 3 个程序项目文件夹，如图 22-2 所示。这些程序项目可以是一般的 Git 程序项目，也就是里面包含“.git”子文件夹，或是 bare 类型的 Git 文档库。如果要开放这 3 个程序项目给其他人读取，只要按照下列步骤操作即可：

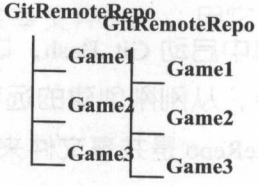


图 22-2　计算机的 D 磁盘中的程序项目

1 步骤　启动 Git Bash 程序。

2 步骤　切换到“D:\ GitRemoteRepo”文件夹。

3 步骤　执行以下指令，其中的“--export-all”选项表示要开放当前文件夹中的所有 Git 文档库。

```
git daemon --export-all
```

4 步骤　网络上的其他计算机可以在 Git Bash 程序中执行以下指令，获取我们开放的程序项目。

```
git clone git://网址或 IP 地址/GitRemoteRepo/Game1
```

如果希望可以独立控制每一个程序项目是否要开放，就不要使用“--export-all”选项，而是在想要开放的程序项目的“.git”子文件夹内，创建一个名为“git-daemon-export-ok”的空文件。如果是 bare 类型的 Git 文档库，就直接把这个文件放在第一层的文件夹即可。

Git Daemon 默认只提供读取的功能。也就是说，其他计算机不能执行“git push”指令更改我们的程序项目。这是为了安全考虑，因为

Git Daemon 没有用户验证的机制，一旦开放写入的功能，任何人都可以修改我们的程序项目。但是，如果我们想要开放 Git Daemon 的写入功能，可以在启动 Git Daemon 的时候，加上“--enable=receive-pack”选项。如此一来，所有对外开放的程序项目就可以被其他人修改。

除了以上介绍的功能之外，还有其他选项可以控制 Git Daemon 的工作方式，以下是一些常用的选项：

1. “--base-path='文件夹路径'”

 这个选项可以简化其他计算机读取程序项目时所指定的路径。如果没有使用这个选项，其他计算机必须在指令中输入完整的程序项目路径，也就是前面范例的“GitRemoteRepo/game1”。如果我们在启动 Git Daemon 时加上如下的选项（如果指定的路径中没有空格，可以省略单引号）：

```
git daemon --export-all --base-path='/d/GitRemoteRepo'
```

 其他人就可以使用以下指令来读取程序项目：

```
git clone git://网址或 IP 地址/Game1
```

2. “--port=通信端口”

 Git Daemon 默认是以 9418 号通信端口来传输数据。如果这个通信端口被防火墙阻挡，可以使用这个选项指定使用其他的通信端口，例如：

```
git daemon --export-all --port=80
```

 如此一来，其他计算机要连接的时候，就要指定相同的通信端口，也就是：

```
git clone git://网址或 IP 地址:通信端口/程序项目路径
```

3. “--verbose”

Git Daemon 运行时，默认只会显示重要的信息。如果想要看到完整的信息，可以在启动 Git Daemon 时加上这个选项。

补充说明

执行 Git Daemon 指令之后，Git Bash 程序的命令行会暂停运行，直到我们同时按下键盘上的 Ctrl 和 C，让 Git Daemon 停止执行。如果希望执行 Git Daemon 的时候，可以继续操作 Git Bash 程序。只要在 Git Daemon 指令最后加上“&”，例如“git daemon --export-all &”。这样 Git Daemon 就会在后台执行，而不会影响 Git Bash 程序的操作。如果要停止后台执行的程序，可以先用“ps”指令查看当前执行中的程序，里面会包含 Git Daemon。记下它的 PID 号码，再执行“kill PID 号码”指令来停止 Git Daemon 程序。



第23课

使用 HTTP/HTTPS 访问 Git Server

绝大部分的公司都不想把自己辛苦开发的程序项目，存储在公司以外的地方。尤其对于高科技公司来说，软件是最重要的财产之一，应该受到严密的保护。因此，为了存储这些重要的知识产权，需要在公司内部创建一个私有的 Git Server，并且严格管制访问权限，以避免发生被窃取的情况。这个单元我们要介绍如何架设可以通过 HTTP 以及 HTTPS 访问的 Git Server，并且加入账号和密码的保护机制。

HTTP 是 WWW 使用的通信协议，它是通过 8080 通信端口来传送数据的。Git 程序本身就支持 HTTP，因此我们只要设置好 Web Server，就可以让外部的 Git 程序通过 HTTP 通信协议和 Web Server 内部的 Git 程序互传数据，图 23-1 是这种运行模式的架构图。我们可以使用 Web

Server 管理和控制账号和密码。只有通过账号和密码验证的人，才能够访问 Web Server 内部的 Git 文档库。这个单元我们就来介绍如何从无到有创建这种运行模式的服务器。

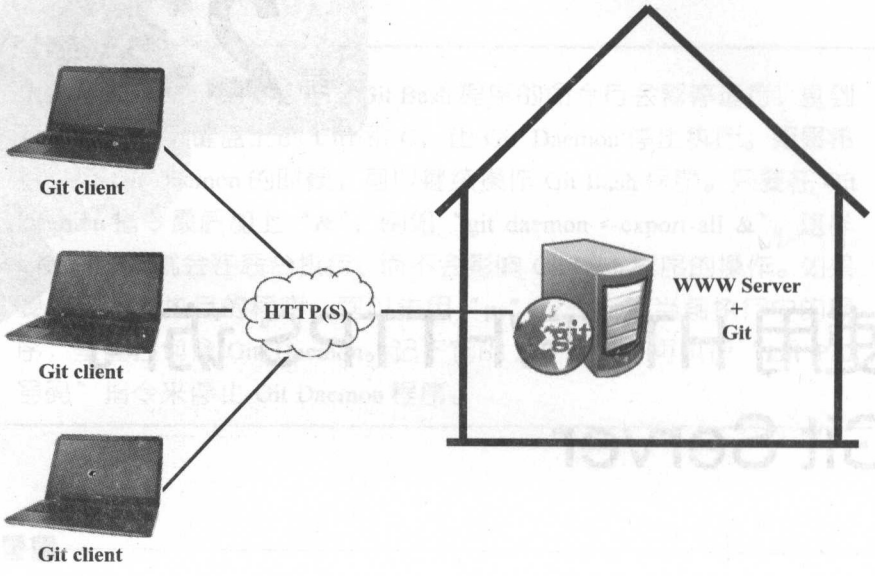


图 23-1　Web Server 搭配 Git 的架构

23-1　使用 HTTP 访问 Git Server

首先我们必须在服务器中安装 Git 程序。Git 程序并没有区分服务器或是客户端专用的版本。也就是说，我们在服务器安装的也是一般的 Git 程序。所以请读者按照第 1 单元的说明，完成 Git 程序的安装。接着是安装 Web Server 软件。我们使用的是最普遍的 Apache 服务器。以下网址提供 Windows 系统使用的最新版本：

http://www.apachelounge.com/download/

下载压缩文件以后，按照 ReadMe.txt 文件的说明，将 Apache 软件包解

压缩到指定的文件夹（如果存储在不同的文件夹，必须自行修改 httpd.conf 文件中的设置）。完成解压缩之后，从“开始菜单 → 附件程序”启动“命令提示符”窗口。再使用“cd”指令切换到 Apache 文件夹里面的 bin 子文件夹，执行 httpd.exe。Windows 系统会显示一个警告窗口，询问是否要让当前执行的程序使用网络功能，请选择允许。现在我们要测试 Apache 的功能是否正常。请开启网页浏览器，在网址栏输入“http://127.0.0.1”。如果网页浏览器的屏幕画面显示出了信息，就表示 Apache 安装成功。

补充说明

执行 httpd.exe 之后，Apache 程序会在后台执行。如果要停止 Apache，先切换到“命令提示符”窗口，同时按下键盘上的 Ctrl 和 C 键。httpd.exe 程序就会停止，“命令提示符”窗口会重新显示提示字符。

接下来要修改 Apache 的设置，让它可以和 Git 程序一起运行，请按照以下步骤进行：

1 步骤 在服务器的磁盘中创建一个存放 Git 文档库的文件夹。以后我们会把所有的程序项目都放在这个文件夹内。为了方便说明，假设我们创建的文件夹路径是“C:\GitRemoteRepo”。

2 步骤 编辑 Apache 的配置文件。它在 Apache 文件夹中的 conf 子文件夹，文件名叫作 httpd.conf。打开配置文件之后，找到“<Directory />”开头的段落。这个段落是用来控制指定的路径是否可以让外部的用户访问。默认是从根目录开始全部不能访问。但是，现在我们要让步骤 1 创建的文件夹和 Git 程序的文件夹，可以让外部用户访问（后续步骤会加上账号和密码的管理和控制）。因此请在原来的“<Directory />...</Directory>”段落后面，加入以下设置：

```
# "C:/GitRemoteRepo"是步骤 1 创建的文件夹路径
<Directory "C:/GitRemoteRepo">
    Options Indexes FollowSymLinks
    AllowOverride None
```

```
    Order allow,deny
    Allow from all
</Directory>

# "C:/Program Files (x86)/Git"是Git程序的安装路径
<Directory "C:/Program Files (x86)/Git/libexec/git-core/">
    Options Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

3 步骤 在同一个配置文件的最后加入以下程序代码。这一段程序代码是启动 Git 程序读取 Git 文档库。GIT_PROJECT_ROOT 是设置 Git 文档库的路径，另外还有 Git 程序的安装路径也要根据实际的情况进行修改。

```
SetEnv GIT_PROJECT_ROOT C:/GitRemoteRepo
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAliasMatch \
        "(?x)^/git/(.*/(HEAD | \
                        info/refs | \
                        objects/(info/[^/]+ | \
                                [0-9a-f]{2}/[0-9a-f]{38} | \
                                pack/pack-[0-9a-f]{40}\.(pack|idx)) | \
                        git-(upload|receive)-pack))$" \
"C:/Program Files (x86)/Git/libexec/git-core/git-http-backend.
exe/$1"
```

4 步骤 继续在同一个配置文件的最后加入以下设置。它的作用是当 HTTP 指定读取“/git”这个路径时，就要使用指定的文件验证账号和密码。我们指定的账号文件就放在 Git 文档库的文件夹中，并且取名为 htpasswd。“/git”这个名称是固定的，不可以改变。Git 程序会自动把“/git”换成 GIT_PROJECT_ROOT 所设置的路径。如果没有加入这一段，任何人都可以读取 Git 文档库。

```
<Location /git>
AuthType Basic
```

```
AuthName "GIT Repository"
AuthUserFile "C:/GitRemoteRepo/htpasswd"
Require valid-user
</Location>
```

5 步骤 现在我们要用 Apache 的工具程序创建账号文件。切换到“命令提示符”窗口，使用“cd”指令，把工作目录切换到 Apache 文件夹里面的 bin 子文件夹，然后执行以下指令：

```
htpasswd -cmb C:/GitRemoteRepo/htpasswd 账号名称 密码
```

6 步骤 启动 Git Bash 程序，切换到步骤 1 创建的文件夹。执行下列指令，创建一个远程 Git 文档库：

```
git init --bare 程序项目名称
```

7 步骤 现在我们要测试是否可以通过 HTTP 获取远程 Git 文档库。由于我们已经修改了 Apache 的配置文件，必须重新启动 httpd.exe，新的设置才会生效。重新启动 httpd.exe 之后，创建一个测试用的文件夹。然后切换到 Git Bash 程序，把工作目录换到这个测试的文件夹。接着执行以下指令，使用 HTTP 获取远程 Git 文档库：

```
git clone http://127.0.0.1/git/程序项目名称
```

程序会显示信息，要求输入账号。请输入步骤 5 设置的账号，再按 Enter 键。接着要求输入密码，同样输入步骤 5 设置的密码。之后就会开始复制远程 Git 文档库。由于当前远程 Git 文档库中没有任何数据，程序会显示以下警告信息，这是正常的情况。

```
warning: You appear to have cloned an empty repository.
```

8 步骤 接着我们在复制得到的程序项目的文件夹内，修改程序项目的文件，再执行 commit。接着执行以下指令，把新的文档库内容送回给远程 Git 文档库。

```
git push -u origin master
```

以上是直接在服务器上测试。确定 Apache 和 Git 都可以正常运行之后，就可以换其他计算机进行测试。后续我们可以在 Apache 的账号文件中加入其他的账号和密码，也可以在步骤 1 创建的 Git 文档库文件夹内创建更多程序项目的文件夹。

23-2 使用 HTTPS 访问 Git Server

HTTP 通信协议在传送数据的时候，不会将数据加密，因此黑客可以看到网络上传送的数据内容。如果换成使用 HTTPS，数据在传送之前会先加密，这样即使黑客截获了网络上传送的数据，也不会知道内容究竟是什么。Apache 本身就支持 HTTPS 通信协议，只要通过适当的设置，就可以使用 HTTPS 来访问 Git Server，使用上会更安全。要使用 HTTPS 通信协议必须完成以下修改：

1. 创建证书。
2. 修改 Apache 的 httpd.conf 文件。
3. 修改 Apache 的 httpd-ssl.conf 文件。

详细的操作步骤如下：

1 步骤 切换到“命令提示符”窗口（注意，必须以系统管理员的方式启动，否则因为权限不足，后续步骤会显示错误信息）。使用“cd”指令把工作目录切换到 Apache 程序的文件夹。

2 步骤 执行以下指令，创建 HTTPS 通信协议使用的证书，按照要求输入密码两次，密码最少包含 4 个字符。

```
bin\openssl genrsa -des3 -out server.key 2048 -config
conf\openssl.cnf
```

3 步骤 执行以下指令，按照提示输入密码和相关信息，请参考下列说明。

```
bin\openssl req -new -key server.key -out server.csr -config
conf\openssl.cnf
Enter pass phrase for server.key:  ←输入步骤 2 设置的密码
.
.(信息说明)
.
-----
Country Name (2 letter code) [AU]:  ←输入国名或地区代码，例如 CN
State or Province Name (full name) [Some-State]:  ←输入国家名称，
例如 China
Locality Name (eg, city) []:  ←输入城市名称，例如 Beijing
Organization Name (eg, company) [Internet Widgits Pty Ltd]:  ←可
省略
Organizational Unit Name (eg, section) []:  ←可省略
Common Name (e.g. server FQDN or YOUR name) []:  ←输入主机网址或是
个人名称，例如 Tom
Email Address []:  ←可省略

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:  ←可省略
An optional company name []:  ←可省略
```

4 步骤 执行以下指令，输入步骤 2 设置的密码。

```
bin\openssl x509 -req -days 3650 -in server.csr -signkey server.key
-out server.crt
```

5 步骤 把 server.key 改名为 server.key.old，然后执行以下指令：

```
bin\openssl rsa -in server.key.old -out server.key
```

6 步骤 将产生的 server.key 和 server.crt 文件移到 Apache 程序文件夹中的 conf 子文件夹。现在我们已经设置好 HTTPS 通信协议所需要的证书。

7 步骤 修改 Apache 的配置文件 httpd.conf ，删除下列 3 行前面的“#”字符。

```
#LoadModule socache_shmcb_module modules/mod_socache_shmcb.so
#LoadModule ssl_module modules/mod_ssl.so
#Include conf/extra/httpd-ssl.conf
```

8 步骤 重新启动 httpd.exe，让新的设置生效。

启用 HTTPS 通信协议之后，我们就可以把前一个小节的“git clone”指令里面的 http 换成 https 来进行测试。但是读者会发现，当我们复制远程 Git 文档库的程序项目时，会显示以下错误信息：

```
SSL certificate problem: self signed certificate
```

因为现在这个网站的 SSL 证书是我们自己生成的，不是由第三方认证机构签发的。而且 Git 程序默认会检查 SSL 证书的合法性，结果就会显示以上的错误信息，并且停止执行。我们可以关闭 Git 这项检查功能，也就是执行指令时，使用“-c”选项，强制关闭 SSL 证书检查的功能，就可以成功复制程序项目。

```
git -c http.sslVerify=false clone https://网址或 IP 地址/git/程序项目名称
```

只是这样做的话，每一次操作远程 Git 文档库的时候，都要额外加上这项设置，因此会觉得比较麻烦。另一种方式是直接在 Git 的配置文件中，把检查 SSL 证书的功能关闭：

```
git config http.sslVerify false
```

以上指令只会更改当前所使用的程序项目，其他程序项目不受影响。设置之后，就可以按照原来的方式操作远程 Git 文档库。



第24课

使用SSH认证和加密的Git Server

SSH是Secure Shell的简称，它可以用来进行用户认证以及对传输的数据加密和压缩。加密后的数据会变成一堆杂乱无章的内容，因此即使在传输的过程中被黑客截取，黑客也无法得知传输的数据究竟是什么。SSH技术能够有效提升网络传输的安全性。Git本身也支持SSH通信协议，可以直接当成SSH Client。不过，我们还是需要一个SSH Server软件，它负责SSH账号管理以及为传输的数据进行加密和压缩的工作。

24-1 SSH 的工作原理

SSH 的运行架构和一般的网络服务类似，一端称为 SSH Client，另一端称为 SSH Server，如图 24-1 所示。比较特别的是，这两端都要创建一组 key，这组 key 称为“密钥”。每一组“密钥”都是由两个 key 配对而成，一个称为“公钥”（public key），另一个称为“私钥”（private key）。“公钥”的用途是给传输的数据加密，也就是让数据变成乱码。“私钥”的用途是把它对应的“公钥”所加密的数据还原（即解密）。如果“公钥”和“私钥”配对错误，就无法成功还原数据。“密钥”必须用特定的工具程序来生成。

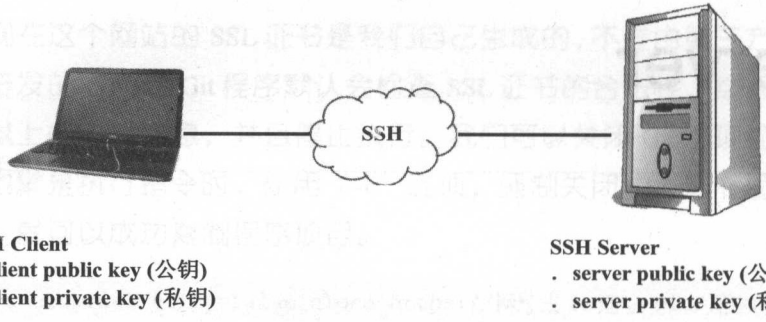


图 24-1　SSH 的运行架构

在 SSH 开始运行之前，必须先把自己的“公钥”交给对方。不管是 Client 还是 Server 都一样。这样，当对方要把数据传送过来的时候，才能够用我们的“公钥”进行加密。当我们收到数据的时候，再用自己的“私钥”进行解密。所以图 24-1 的 SSH Client 必须把它的“公钥”传给 SSH Server。同样地，SSH Server 也必须把它的“公钥”传给 SSH Client，最后变成图 24-2 所示的状态。了解 SSH 如何运行之后，接下来就开始介绍如何安装。

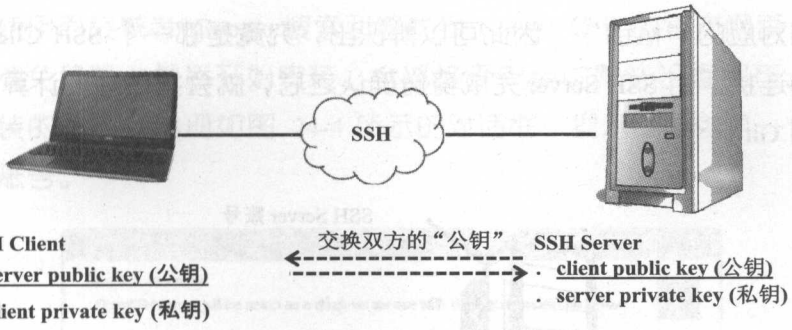


图 24-2　SSH 运行之前必须先把“公钥”传给对方

补充说明

图 24-1 和图 24-2 是简化后的结果。其实 SSH 的“密钥”并不是以计算机为单位，而是以用户账号来区分的。也就是说，同一台计算机上的用户都可以创建自己的“密钥”。当不同用户登录计算机与 SSH Server 连接时，都是使用不同的“密钥”。

24-2　安装和设置 Copssh Server

Copssh 是在 Windows 平台上运行的 SSH Server，读者可以使用百度或者 Google 搜索，就可以找到它的官方网站。它分成付费版和免费版两种。免费版只能够创建一个 SSH 用户账号，付费版则没有限制。我们只需要下载免费版即可，因为 Git Server 只需要使用一个 SSH 账号。为了让读者了解 SSH Server 和 Git 如何搭配运行，我们用图 24-3 来解释。经过前面的说明，我们已经知道 SSH Server 必须存储 SSH Client 的“私钥”。但是 SSH Server 是在 Windows 系统中运行，我们必须能够让外部计算机先通过 Windows 系统，才能到达 SSH Server。也就是说，SSH Server 必须与一个 Windows 系统的账号连接（通常我们把这个 Windows 账号取名为“git”）。当外部计算机使用这个账号登录时，就会进入 SSH Server。在 SSH Server 中，我们存储所有 SSH Client 的“私钥”。这样 SSH Server 就可以根据外部计算机传送过来的 SSH 数据，

找到对应的“私钥”，因此可以辨识出，究竟是哪一个 SSH Client 在创建连接。当 SSH Server 完成身份确认之后，就会执行外部计算机要求的 Git 指令。

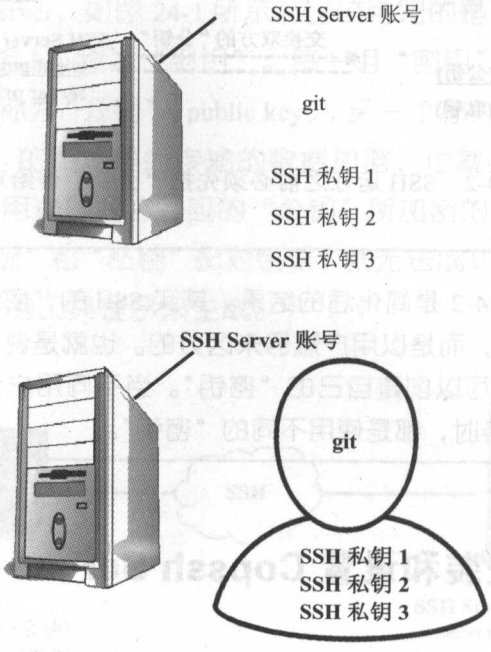


图 24-3　SSH Server 和 Git 的运行模式

现在读者应该对于 Git 如何搭配 SSH 运行有了基本的概念，这将有助于了解接下来的安装和设置的过程，现在我们就可以开始动手了。

1 步骤 首先必须在 Windows 系统新增一个名为“git”的账号。这个账号不需要管理员的权限，只要给它一般用户的权限即可。我们可以从“控制面板 > 系统管理工具 > 计算机管理”来启动计算机管理程序。然后从左边的选项展开“本机用户和群组 > 用户”就可以执行新增用户的操作。

2 步骤 使用百度或者 Google 搜索引擎找到 Copssh 软件的官方网页，下载免费版，然后开始安装。全部接受安装的默认设置即可。安装的过程会出现如图 24-4 所示的对话框，提示必须创建一个新账号。

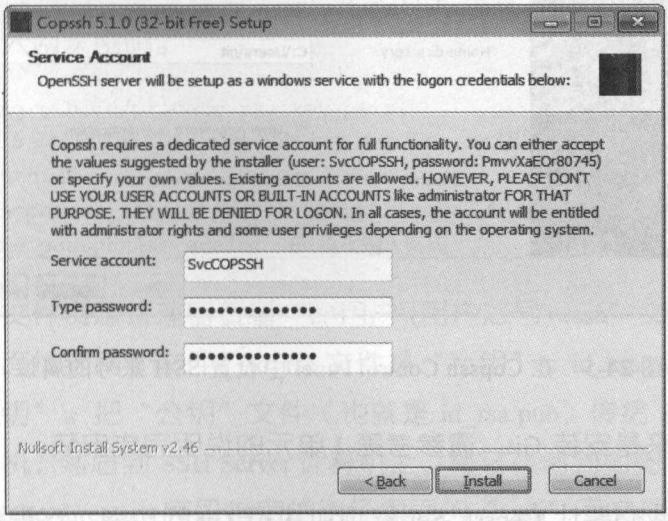


图 24-4　安装 Copssh 时需要创建一个新账号

补充说明

Copssh 其实会在 Windows 系统创建两个账号，除了图 24-4 所示的账号之外，还会创建一个称为 sshd 的账号。

3 步骤 执行 Copssh Control Panel，屏幕画面会显示 Service is running 信息。单击左上角的 Users 标签页，再单击窗口右边的 Add 按钮。在操作的过程中，选择步骤 1 创建的 git 账号，然后设置如图 24-5 所示的属性。

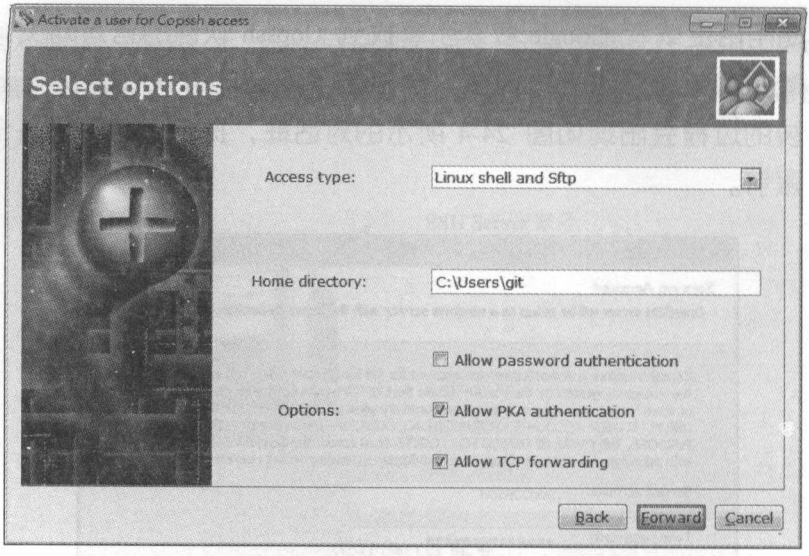


图 24-5　在 Copssh Control Panel 中设置 SSH 账号的属性

4 步骤 接下来是安装 Git，请参考第 1 单元的说明完成安装。

5 步骤 现在我们要让 Copssh Server 可以执行 Git 的功能。这需要将几个 Git 执行文件和相关的链接库复制到 Copssh 文件夹中的 bin 子文件夹（提示：不可以使用创建链接的方式，必须确实把文件复制过来，否则执行 Git 会出现错误）。这些文件的位置如下，请使用 Windows 文件资源管理器找到这些文件，将它们复制到 Copssh 文件夹中的 bin 子数据。

1. (Git 安装文件夹)\bin\git.ext
2. (Git 安装文件夹)\bin\libiconv-2.dll
3. (Git 安装文件夹)\libexec\git-core\git-receive-pack.exe
4. (Git 安装文件夹)\libexec\git-core\git-upload-archive.exe
5. (Git 安装文件夹)\libexec\git-core\git-upload-pack.exe

6 步骤 接下来要在 SSH Client 端创建“密钥”。在 SSH Client 端的计算机上执行 Git Bash，然后输入以下指令：

```
ssh-keygen
```

程序会询问存储“密钥”文件的路径，直接按 Enter 键即可，然后输入两次密码。

```
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/(用户账号)/.ssh/id_rsa):
←直接按 Enter 键
Enter passphrase (empty for no passphrase): ← 输入密码
Enter same passphrase again: ← 再次输入密码
```

使用文件资源管理器查看“c:\用户\(用户账号)\.ssh”文件夹，会在里面找到两个文件：id_rsa 文件是“私钥”；id_rsa.pub 文件是“公钥”。把“公钥”文件（也就是 id_rsa.pub）传给 SSH Server 计算机，然后到 SSH Server 计算机上，将它存储到如图 24-5 所示的 Home directory 字段指定的路径里面的“.ssh”子文件夹中，并且改名为 authorized_keys。

补充说明

后续如果要加入其他 SSH Client 传过来的“公钥”，可以先打开 Copssh Control Panel，单击左上角的 Users 标签页。找到程序屏幕画面下方的 Keys 按钮。单击该按钮会显示出如图 24-6 所示的对话框。单击其中的 Import 按钮，会显示出另一个编辑窗口。我们先用 Windows“记事本”程序来打开 SSH Client 传过来的“公钥”文件，复制其中的内容，再把它粘贴到这个编辑窗口。单击 Apply 按钮，回到如图 24-6 所示的屏幕画面，就会新增一个 Public key 项。如果要删除“公钥”，先单击要删除的项，然后单击 Delete 按钮。

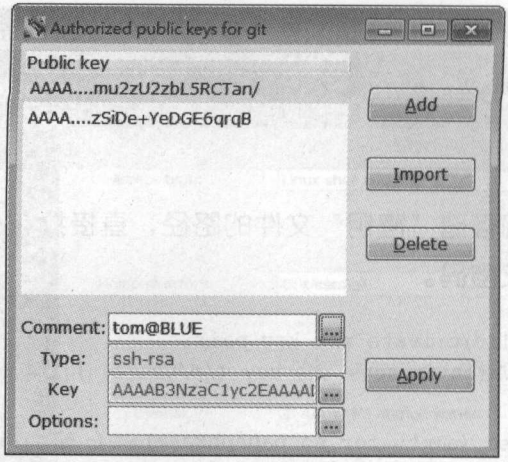


图 24-6　Copssh Control Panel 程序加入 SSH Client“公钥”的操作界面

7 步骤 回到 Copssh Control Panel 程序。单击左上角的 Status 标签页，再单击屏幕画面上的绿色按钮，停止 SSH Server。等到按钮变成红色之后，再按一次，让它重新启动，这样新的设置才会生效。屏幕画面最下方的按钮可用于查看 SSH Server 产生的 log 信息。

8 步骤 现在我们要测试 SSH Server 是否可以正常运行。在 SSH Client 端的计算机上执行 Git Bash 程序，输入以下指令：

```
ssh git@SSH Server 的网址或 IP 地址
```

程序会要求输入密码，请输入我们创建“密钥”时设置的密码。如果成功和 SSH Server 创建连接，屏幕画面会显示如下信息：

```
$ ssh git@SSH Server 的网址或 IP 地址
Enter passphrase for key '/c/Users/用户账号/.ssh/id_rsa': ← 输入密码
Last login: 日期和时间 from IP 地址

git@SSH Server 计算机名称 ~   ← 状态栏显示已经登录 SSH Server
$
```

如果要结束与 SSH Server 的连接，可以执行 exit 指令。

9 步骤 接着要测试 SSH Server 的 Git 功能。换到 SSH Server 的计算机，启动 Git Bash 程序，切换到如图 24-5 中所示的 Home directory 字段中指定路径的文件夹中。执行下列指令，创建一个 Bare 类型的 Git 文档库：

```
git init --bare 程序项目名称
```

10 步骤 回到 SSH Client 端的计算机，切换到 Git Bash 程序，输入以下指令：

```
git clone git@SSH Server 的网址或 IP 地址:程序项目名称
```

程序会要求输入密码，请输入我们创建“密钥”时的密码。如果执行成功，屏幕画面会显示如下信息：

```
$ git clone git@SSH Server 的网址或 IP 地址:程序项目名称
Cloning into '程序项目名称'...
Enter passphrase for key '/c/Users/用户账号/.ssh/id_rsa': ← 输入密码
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

11 步骤 最后是测试上传 Git 文档库的功能。先修改程序项目文件夹中的文件，然后执行 commit，接着执行以下指令，把新的文档库内容送回给远程 Git 文档库。

```
$ git push -u origin master   ← 执行这个指令，把新的文档库内容送回去
Enter passphrase for key '/c/Users/用户账号/.ssh/id_rsa': ← 输入密码
Counting objects: …, done.
Writing objects: 100% (…), …, done.
Total …, reused …
To git@SSH Server 的网址或 IP 地址:程序项目名称
 * [new branch]      master -> master
```

如果以上测试都没有问题，就一切大功告成！如果执行步骤 8 的 SSH 连接出现错误，可以尝试停止 SSH Server 再重新启动。如果在步

骤 8 之后的测试出现问题，就是属于 Git 部分的错误。可以先尝试在本机进行 Git 操作测试，找出问题点，并加以解决，然后再进行远程操作测试。

最后再做一些补充说明。以上的安装步骤并没有让 Copssh 生成“密钥”，也没有把 Copssh 的“公钥”传给 SSH Client，看起来和我们之前讨论的不太一样？其实这是因为 SSH Server 会自动完成这些步骤，如以下说明：

1. 当 Copssh 第一次启动的时候，会自动生成“密钥”，并存储在它的 etc 子文件夹内。
2. 当 SSH Client 要求连接的时候，如果通过身份验证，Copssh 就会把自己的“公钥”，传给 SSH Client。
3. SSH Client 收到 Copssh 的“公钥”之后，会把它存储在 c:\Users\(用户账号)\.ssh 文件夹里面的 known_hosts 文件中。

完成以上步骤之后，双方就可以开始互传数据了。



附录 A

Git 常用指令一览表

Git 总共提供了超过一百个以上的指令让我们执行各种操作(可以使用“git help -a”指令显示完整的指令列表），每一个指令又有许多选项可以搭配，但是我们不可能，也不需要完全记住这些指令。就实用性而言，只要熟练部分指令和选项的用法，就可以满足一般应用的需求。以下我们将常用的 Git 指令，按照字母顺序列表说明，以方便读者查阅。

Git 指令（按照字母顺序排序）	功能说明
git add .	将全部文件的内容加到 Git 索引，以便执行 commit。这个指令不会检查文件夹中是否有文件被删除。要注意的是，只有执行“git add”时的文件内容会被加入 Git 索引。如果后来又修改了文件，新的文件内容不会在 Git 索引中。我们必须重新执行“git add”指令，才会更新 Git 索引。
git add 文件名 文件名 …	将指定的文件的内容加到 Git 索引，以便执行 commit 操作。要注意的是，只有执行“git add”时，文件内容才会被加入 Git 索引。如果后来又修改了文件，新的文件内容不会在 Git 索引中。我们必须重新执行“git add”指令，才会更新 Git 索引。
git add -A	除了把全部文件的内容加到 Git 索引以外，也会检查文件夹中是否有文件被删除。这些被删除的文件会标记在 Git 索引中，当执行 commit 指令的时候，被标记删除的文件也会从新的 commit 节点中被删除。我们可以从 Git 文档库的历史版本中找回被删除的文件。
git add --update 或是 git add -u	对比当前文件夹中的文件内容和 Git 文档库中的文件内容，把有修改的部分和删除的文件加到 Git 索引，以便执行 commit。这个指令不会增加新的文件到 Git 索引，只会更新或是删除文件。
git blame 文件名 或是 git blame -L 起始行,结束行 文件名 或是 git blame -L 起始行, 文件名 或是 git blame -L,结束行 文件名	显示文件的每一行是由谁修改。可以搭配“-L”选项，指定要从哪一行开始到哪一行结束。如果没有指定起始行，表示是从文件的第一行开始。如果没有指定结束行，表示要到文件的最后一行。
git branch 自己取的分支名称 \ commit 节点标识符或是标签	按照参数的多少，会有不同的功能： 1. 如果最后指定了 commit 节点标识符或是标签，就会从该节点“长”出分支；如果没有指定 commit 节点，就会从最新的 commit “长”出分支； 2.“git branch”指令后面没有接任何参数时，会列出当前文档库中正在开发的所有分支。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git branch 新分支的名称 已经存在的分支	从特定的分支，再长出另一个新的分支。
git branch -a	列出文档库和远程文档库中所有的分支。
git branch -d 要删除的分支名称	删除指定的分支。必须先切换到另一个分支，才能执行这个指令。
git branch -D 要删除的分支名称	在一般情况下，分支应该先合并到另一个分支，之后才能够被删除。如果我们要删除还没有合并的分支，Git 会显示错误信息，并且停止删除分支的操作。如果确定要删除还没有合并的分支，可以使用“-D”选项，要求 Git 强制执行删除分支的操作。
git branch --list 分支名称样板	显示符合“分支名称样板”的所有分支，例如以下指令范例会显示所有以“bug/”开头的分支： git branch --list bug/*
git branch -m 新的分支名称	更改分支的名称。必须先切换到该分支，才能够执行这个指令。
git checkout 文件 1 文件 2 … 或是 git checkout .	Git 会先找索引中有没有该文件，如果有就把它取出；如果没有，就从最新的 commit 节点开始，按照时间顺序往前寻找，然后取出第一个找到的文件版本，每一个文件都用同样的方式处理。 如果要取出文档库中全部文件的最新版本，可以执行“git checkout .”。
git checkout commit 节点标识符或标签 \ 文件 1 文件 2 …	从 Git 文档库的 commit 节点取出指定的文件。如果取出的文件和当前文档库中最新 commit 的文件内容不同，这个取出的文件内容会自动记录在 Git 索引中。下次执行“git commit”指令时，这个取出的文件内容就会存入文档库中成为新的版本。如果要避免这种情况发生，可以在执行“git checkout”指令之后，立刻执行“git reset HEAD”来清除 Git 索引。
git checkout 分支名称	将当前操作的分支切换到指定的分支。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git checkout -f 分支名称	在切换分支的时候，Git 会先对比文档库中当前分支的文件内容是否和将要切换过去的分支的文件内容相同。针对内容不一样的文件，Git 需要从文档库中取出该文件，这是为了让文件夹中的文件符合分支原来的状态。但是为了避免数据遗失，当 Git 要覆盖文件夹中的文件时，会检查该文件的内容是否已经加入文档库。如果还没有加入，Git 会显示警告信息，并且停止执行，以免资料遗失。如果我们确定不想保留这些已经修改却还没有加入文档库的文件，可以加入“-f”选项，这样 Git 就会强制覆盖修改后的文件。
git checkout -b 新分支的名称 \ commit 节点标识符或是标签	创建指定的分支，然后切换到新创建的分支。这个指令等同于先执行“git branch 新分支的名称 commit 节点标识符或是标签”，接着再执行“git checkout 新分支的名称”。如果最后指定了 commit 节点标识符或是标签，就会从该节点“长”出分支。如果没有指定 commit 节点，就会从最新的 commit“长”出分支。
git cherry-pick -n commit 节点标识符或标签	把指定的commit节点的文件版本合并到文件夹中的文件。在默认情况下，执行这个指令会创建一个新的 commit 节点。如果不想要创建新节点，则可以加上“-n”选项。执行这个指令之前，文件夹中被修改的文件必须先存入 Git 文档库，否则会出现警告信息，并且停止执行。
git clone “远程 Git 文档库”的路径 \ “本地 Git 文档库”文件夹名称 或是 git clone \ //(计算机名称)/(“远程 Git 文档库”的路径) \ “本地 Git 文档库”文件夹名称 或是 git clone \ http://(Web 服务器网址或是 IP 地址)/(“远程 Git 文档库”的路径) \ “本地 Git 文档库”文件夹名称 或是 git clone \(Git 账号)@(SSH 服务器的网址或 IP 地址):(“远程 Git 文档库”的路径) “本地 Git 文档库”文件夹名称	从“远程 Git 文档库”复制一个“本地 Git 文档库”到我们的计算机。 第一种方式适用的情况是“远程 Git 文档库”在我们自己的计算机上。 第二种方式适用的情况是“远程 Git 文档库”在局域网的计算机上，并且以“共享文件夹”的方式共享。 第三种方式适用的情况是“远程 Git 文档库”在 Web 服务器上，并且已经设置好 Web 服务器，让它能够执行 Git。 第四种方式适用的情况是“远程 Git 文档库”在 SSH 服务器上，并且已经创建了一个 Git 程序专用的账号。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git clone --bare \ 程序项目文件夹名称 \ “远程 Git 文档库”的路径	从“本地 Git 文档库”复制出 Bare 类型的“远程 Git 文档库”。我们通常会帮 Bare 类型的 Git 文档库加上扩展名“.git”，例如 game.git。
git commit -m ‘这次操作的说明’ \ --author=‘操作者姓名 <email 邮箱>’	把当前 Git 索引的内容送进文档库存储。每一次执行 commit 一定要附加说明和操作者信息。如果没有使用“-m”选项，Git 会启动文字编辑程序让我们输入操作说明。默认的文字编辑程序是 vi，我们可以使用“git config”指令设置其他的文字编辑程序，详细操作方式请参考第 2 单元的说明。 除了使用“--author”选项输入操作者信息以外，也可以把操作者的信息记录在 Git 配置文件中，这样就不用加上“--author”选项，详细操作方式请参考第 2 单元的说明。
git commit -a -m ‘这次操作的说明’ \ --author=‘操作者姓名 <email 邮箱>’ 或是 git commit --all -m ‘这次操作的说明’ \ --author=‘操作者姓名 <email 邮箱>’	这个指令的效果等同于先执行“git add -u”再执行“git commit -m ‘这次操作的说明’ --author=‘操作者姓名 <email 邮箱>’”。 首先对比当前文件夹中的文件内容和 Git 文档库中的文件内容，把有修改的部分和删除的文件加到 Git 索引，然后执行 commit。这个指令不会把新的文件加到 Git 文档库。
git commit --amend -m ‘新的操作说明’ \ --author=‘操作者姓名 <email 邮箱>’	修改最近一次 commit 节点的操作说明，或是操作者信息据。
git config –l 或是 git config --global -l 或是 git config --system -l	只有使用“-l”选项时会显示三个不同层级的配置文件中所有的设置项。低优先权配置文件的设置项会先显示，最高优先权配置文件的设置项显示在最后。 加入“--global”选项时会显示中优先权配置文件中的设置项，也就是登录账号的 home directory 里面的.gitconfig 文件中的设置。 加入“--system”选项时会显示最低优先权配置文件中的设置项，也就是 Git 程序安装文件夹里面的 etc\gitconfig 文件中的设置。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git config 设置项名称 ‘设置值’ 或是 git config --global 设置项名称 ‘设置值’ 或是 git config --system 设置项名称 ‘设置值’	在 Git 配置文件中加入或是修改设置。如果“设置值”中没有空格，可以省略单引号。 如果没有使用任何选项，表示要将设置项写到当前操作中的 Git 文档库中的配置文件。 如果加入“--global”选项，表示要将设置项写到登录账号的 home directory 里面的.gitconfig 配置文件中。 如果加入“--system”选项，表示要将设置项写到 Git 程序安装文件夹里面的 etc\gitconfig 配置文件中。
git config alias.指令别名 \ ‘正式的指令和选项’ 或是 git config –global alias.指令别名 \ ‘正式的指令和选项’ 或是 git config –system alias.指令别名 \ ‘正式的指令和选项’	定义指令的“别名”（alias），也就是用简短的缩写来表示某一个指令。“指令别名”中不可以包含空格，如果“正式的指令和选项”里面没有空格，可以省略单引号。 如果没有使用任何选项，表示要将设置项写到当前操作中的 Git 文档库中的配置文件。 如果加入“–global”选项，表示要将设置项写到登录账号的 home directory 里面的.gitconfig 配置文件中。 如果加入“–system”选项，表示要将设置项写到 Git 程序安装文件夹里面的 etc\gitconfig 配置文件中。
Git config –unset 设置项名称 或是 git config –global –unset 设置项名称 或是 git config –system –unset 设置项名称	删除 Git 配置文件中某一个设置项。 如果没有使用任何选项，表示要修改当前操作中的 Git 文档库中的配置文件。 如果加入“–global”选项，表示要修改登录账号的 home directory 里面的.gitconfig 配置文件。 如果加入“–system”选项，表示要修改 Git 程序安装文件夹里面的 etc\gitconfig 配置文件。

（续表）

Git 指令（按照字母顺序排序）	功能说明
Git diff 文件名 或是 git diff commit 节点 1 commit 节点 2 文件名 或是 git diff –no-index 文件 1 文件 2 或是 git diff –cached 文件名 或是 git diff commit 节点 文件名	使用 Git 内建的文件对比程序，比较文件夹、Git 索引和 Git 文档库中各种版本的文件内容。不同的格式和选项是用来指定不同的文件版本，详细的说明及范例请参考第 4 单元的介绍。
git difftool 文件名 或是 git difftool commit 节点 1 commit 节点 2 文件名 或是 git difftool —no-index 文件 1 文件 2 或是 git difftool —cached 文件名 或是 git difftool commit 节点 文件名	使用外部文件对比程序，比较文件夹、Git 索引和 Git 文档库中各种版本的文件内容。不同的格式和选项是用来指定不同的文件版本，详细的说明及范例请参考第 4 单元的介绍。设置外部文件对比程序的方法请参考第 2 单元的说明。
Git fetch 或是 git fetch —all	从“远程 Git 文档库”取回当前所在分支的最新数据。完成这项工作之后，我们计算机上的“远程 Git 文档库”的当前分支状态就和实际的情况一致。如果想要一次获取全部分支最新的数据，可以加上“—all”选项。
Git gc —选项	清理 Git 文档库，常用的搭配选项如下： 1. “—aggressive” Git 在默认情况下，会用比较快速的方式检查文档库并完成清理。如果加入这个选项，Git 会用比较仔细的方式检查以及清理，但是需要比较久的时间。这个选项只需要“偶尔”使用即可，太常使用只是浪费时间，不会有明显的帮助。 2. “—auto” 如果加入这个选项，Git 会先判断文档库是否需要清理。如果情况还算良好，就不会执行清理的操作。 3. “—no-prune” 这个选项是要求 Git 不要“清除”文档库中不会用到的数据，只要“整理”它们即可。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git grep ‘要找的字符串’ commit 节点	搜索指定的 commit 节点中的所有文件，然后列出包含该字符串的每一行。如果没有指定 commit 节点，表示要搜索文件夹中的所有文件。 这个指令默认会区分英文大小写。如果不要区分英文大小写，可以加上“-i”选项。如果只要列出包含该字符串的文件，可以加上“-l”选项。如果还想知道每一个文件中有几行含有该字符串，可以换成使用“-c”选项。
Git grep –e ‘要找的字符串 1’ \ -e ‘要找的字符串 2’ commit 节点	以 or 的方式结合要搜索的字符串，也就是说只要出现其中一个就算符合条件。
Git grep –e ‘要找的字符串 1’ –and \ -e ‘要找的字符串 2’ commit 节点	以 and 的方式结合要搜索的字符串，也就是说必须所有的字符串都出现，才算符合条件。
Git init	在当前的文件夹创建一个 Git 文档库。如果这个文件夹已经有 Git 文档库，这个指令就不会再重新创建，也不会修改其中的内容。 Git 文档库其实是名称叫作“.git”的子文件夹，默认它会被隐藏起来，我们可以改变文件夹的查看选项让它显示出来。如果删除这个子文件夹，Git 文档库的内容就会全部消失。
Git init –bare Git 文档库文件夹名称	创建 Bare 类型的 Git 文档库。我们通常会帮 Bare 类型的 Git 文档库加上扩展名“.git”，例如 game.git。
git log	按照时间顺序，从最近一次的 commit 开始，往前列出每一次 commit 的信息，包括标识符、执行人、日期和时间以及说明。
Git log 文件名 1 文件名 2 …	只显示有更改指定文件的 commit。文件的更改包括：加入、删除以及修改内容。
Git log –after=’公元年-月-日 时间’ \ --before=’公元年-月-日 时间’	指定要显示某一段时间期间的 commit 的信息。“–after”可以换成“–since”，“–before”可以换成“–until”。
Git log –author=‘人名’	只显示特定人的 commit 节点信息。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git log --graph --oneline --all --decorate	加上“--graph”选项会用文本模式排列出 commit 节点的演进图。加上“--oneline”选项会用最精简的方式显示。 加上“--all”选项会显示所有分支的 commit 信息。 加上“--decorate”选项表示要标示分支的名称。
git log --stat 或 git log --shortstat 或 git log --numstat	显示每一个 commit 更改程序代码和文件的情况，包括有多少文件被修改了、增加了几行程序代码和删除了几行程序代码。
git ls-files	列出当前 Git 文档库中的文件列表。
git ls-remote	列出“本地 Git 文档库”对应的所有“远程 Git 文档库”。
git merge 分支名称	把指定的分支合并到当前所在的分支。
git merge --abort	合并的过程发生冲突之后，执行这个指令可以放弃合并。Git 文档库和文件夹中的文件内容都会恢复到未执行合并前的状态。
git merge --no-ff 分支名称	“--no-ff”选项表示不要使用 fast-forward merge。
git mergetool	启动 Git 配置文件中指定的 Merge Tool。如果合并分支的时候发生冲突，而且我们已经在 Git 配置文件中完成 mergetool 相关项的设置，就可以执行这个指令，使用外部程序来编辑冲突的文件。
git mv 原来的文件名 新文件名 或是 git mv 原来的文件夹名称 新文件夹名称	更改文件夹中的文件名，或是子文件夹名称，然后把它记录在 Git 索引。接着只要执行“git commit”指令，就可以将更改存入文档库。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git pull 或 git pull --all	*“git pull”指令会执行两项工作：* 1. 从“远程 Git 文档库”取回当前所在分支的最新数据。完成这项工作之后，计算机上的“远程 Git 文档库”的当前分支状态就和实际的情况一致。如果想要一次获取全部分支最新的数据，可以加上“--all”选项。 2. 把“远程 Git 文档库”的分支合并到“本地 Git 文档库”的分支。
git pull --rebase 或 git pull -r	把“git pull”指令的第二个步骤换成“git rebase”（原来是“git merge”）。
git push	执行这个指令时，屏幕画面会显示一段信息，提示我们需要在 Git 的配置文件中加入 push.default 的设置。我们可以执行下列指令，指定 push.default 为 matching： git config --global push.default matching 这样的话，如果只下达“git push”指令，后面没有加上任何参数，Git 会把“本地 Git 文档库”中所有曾经传送给“远程 Git 文档库”的分支，都一并更新。或者，可以执行下列指令，指定 push.default 为 simple： git config --global push.default simple 这样的话，如果只下达“git push”指令，后面没有加上任何参数。Git 会检查当前所在的分支是否在配置文件中记录了它对应的“远程 Git 文档库”的分支。如果有，就会传送更新给“远程 Git 文档库”。否则，就不会执行更新。和 matching 的设置相比之下，simple 是比较安全的做法。因为它只会更新当前所在的分支，不会一次更新全部的分支。
git push origin 分支名称	把指定分支的最新状态送到 origin 属性所对应的“远程 Git 文档库”。执行这个指令不会在配置文件中记录“本地 Git 文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git push --set-upstream origin 分支名称 或 git push -u origin 分支名称	把指定分支的最新状态送到“远程 Git 文档库”。执行这个指令会在配置文件中记录“本地 Git 文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。
git push “远程 Git 文档库”的 url 分支名称	把指定分支的最新状态送到指定的“远程 Git 文档库”。执行这个指令不会在配置文件中记录“本地 Git 文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。
git push --set-upstream \ “远程 Git 文档库”的 url 分支名称 或 git push -u \ “远程 Git 文档库”的 url 分支名称	把指定分支的最新状态送到指定的“远程 Git 文档库”。执行这个指令会在配置文件中记录“本地 Git 文档库”的分支和“远程 Git 文档库”的分支之间的对应关系。
git push --all	把“本地 Git 文档库”中所有的分支传送到“远程 Git 文档库”。
git push “远程 Git 文档库”的名称 --delete \ 分支名称	删除“远程 Git 文档库”中的指定分支。
git rebase 分支名称	把指定分支的修改运用到当前的分支。当前分支会变成从指定的分支的 HEAD 节点长出来。
git rebase --abort	如果执行 rebase 指令后出现冲突的情况，可以使用这个指令取消 rebase 的操作。Git 文档库会恢复到还没有执行 rebase 之前的状态。
git rebase --continue	执行 rebase 指令后出现冲突的情况，而且我们已经编辑好发生冲突的文件，接着就可以执行“git add”指令，把新的文件内容加入 Git 索引，最后再执行这个指令，完成 rebase 的操作。
git reflog HEAD 或是任何分支的名称	显示 HEAD 或是任何分支变动的历史记录。如果不加任何参数，默认会列出 HEAD 变动的历史记录。
git remote -v	显示和“远程 Git 文档库”相关的设置。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git remote add \ “远程 Git 文档库”的名称 \ “远程 Git 文档库”的 url	在“本地 Git 文档库”的配置文件中加入指定的“远程 Git 文档库”名称，并且设置它的 url。
git remote rm “远程 Git 文档库”的名称 或 git remote remove “远程 Git 文档库”的名称	删除“本地 Git 文档库”的配置文件中指定的“远程 Git 文档库”名称。一旦删除“远程 Git 文档库”的名称，所有属于它的追踪分支也会一并消失。如果要再还原回来，只要再执行“git remote add”指令和“git remote update”指令即可。
git remote rename 旧名称 新名称	改变“远程 Git 文档库”的名称。改变“远程 Git 文档库”的名称之后，和它相关的远程追踪分支的名称也会自动更新。
git remote set-url \ “远程 Git 文档库”的名称 新的 url	改变“远程 Git 文档库”的 url。
git remote show “远程 Git 文档库”的名称	显示指定的“远程 Git 文档库”的详细信息。
git remote update	让 Git 对比“本地 Git 文档库”和“远程 Git 文档库”的分支名称，找出对应的分支，并且在“本地 Git 文档库”中创建追踪分支。
git reset HEAD 文件名	将指定文件的内容从 Git 索引中删除。如果没有加上文件名，则会清除 Git 索引中所有的内容。
git reset --soft commit 节点标识符或是标签 或 git reset --mixed commit 节点标识符或是标签 或 git reset --hard commit 节点标识符或是标签	将 Git 文档库“消磁”，让 Git 文档库恢复到某一个 commit 节点的状态，如果使用“--soft”选项，表示只有文档库中的数据会更改，Git 索引和文件夹中的文件都不会受到影响。如果使用“--mixed”选项（这是默认的选项），表示 Git 索引也会恢复到指定节点的状态，但是文件夹中的文件仍然不会受到影响。如果使用“--hard”选项，则文档库、Git 索引和文件夹中的文件都会恢复到指定节点的状态。
git revert commit 节点	回到指定的 commit 节点的前一个节点的文件状态。执行完毕后会新增一个 commit 节点。请注意和“git reset”指令的差别。
git revert --abort	如果执行“git revert”指令的时候发生冲突，可以执行这个指令取消 revert 的操作。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git rm 文件名	Git 会执行两项检查： 1. Git 索引中有没有该文件的内容（也就是刚刚有没有执行过“git add 文件名”），如果有，表示这个文件的内容和文档库中的不一样，为了避免遗失数据，Git 会显示提醒信息，然后放弃执行； 2. 文件夹中的文件内容是不是和文档库中的一样，如果不一样，Git 同样会显示提醒信息，然后放弃执行，以免遗失数据。 如果通过以上两项检查，Git 会马上删除文件夹中的文件，然后在索引中记录要从文档库中删除该文件。最后必须再执行“git commit”指令才会真正从文档库中删除文件。
git rm --cached 文件名	把指定的文件从 tracked 状态变成 untracked 状态，也就是说从此以后不需要在文档库中更新这个文件，而且 Git 索引中这个文件的内容也会被删除，但是 Git 不会从文件夹中删除这个文件（这是加上“--cached”选项最大的差别）。
git shortlog	按照人名的字母顺序，列出每一个人执行 commit 的次数和说明。 加上“--numbered”选项（或是“-n”）可以按照 commit 次数，由高至低按序排列。 如果不需要显示 commit 说明，可以加上“--summary”选项（或是“-s”）。
git show commit 节点标识符或是标签	显示特定 commit 节点的详细信息。commit 节点标识符是一组很长的 16 进制数字，指定 commit 节点标识符时，不需要将它完整列出。一般只要使用最前面 4 个数字即可，Git 会自动找出对应的节点。如果找到超过一个以上的节点，Git 会显示错误信息，这时候换长一点的数字就可以解决。
git show 文件名	显示指定文件最新版本的修改情况。也就是比较文件最新版本和前一个版本的差异。
git show commit 节点:文件名	显示指定的 commit 节点中的特定文件内容。
git stash list	显示 Git 暂存区的状态。

（续表）

Git 指令（按照字母顺序排序）	功能说明
git stash pop 或 git stash apply	取出暂存区的文件，将它们的内容合并到当前文件夹中的文件。
git stash save	这个指令会执行下列两项工作： 存储文件夹中被 Git 追踪的文件和文档库中最新文件版本的差异； 把文件夹中被 Git 追踪的文件还原成文档库中最新的文件版本。
git status	这个指令会执行以下 3 项工作： 1. 检查 Git 索引的内容，看看是否需要执行 commit 存入文档库。 2. 对比文件夹中的文件和文档库中的文件，列出修改过的文件列表。 3. 列出 untracked 状态的文件。
git tag 自定义的标签名称 \ commit 节点标识符或是标签	指定的 commit 节点贴上自定义的标签，之后就可以用这个自定义的标签来指定这个 commit 节点。
git tag -d commit 节点标签	删除自定义的 commit 节点标签。
gitk --all	启动图形操作模式，如果加入“--all”选项，表示要显示全部的分支，否则只会显示当前操作中的分支。

完全学会 Git GitHub Git Server 的 24 堂课

Git杰出的优点让越来越多的项目开发者和管理人员以及公司都开始采用它，因而它的市场占比在快速增加，目前几乎所有的集成开发工具都支持Git，因此熟练掌握Git已经成为业内项目开发者和管理人员必备的技能。

本书以循序渐进的方式，先从单机操作模式开始，让读者熟悉Git的基本用法，接着学习Git在团队开发模式中的实际运用，并通过介绍GitHub、Bitbucket、GitLab等提供Git服务的网站，让读者可以和全世界的项目开发人员交流，提升程序开发的经验和能力。

本书的最后介绍了架设Git Server的几种方法，让每一个项目开发团队，可以按照自己的软硬件环境，选择适合的方式来有效管控自己开发的项目。

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

ISBN 978-7-302-43779-6
9 787302 437796 >
定价：49.00元